Copyrighted Material

USB Design by Example

A Practical Guide to Building I/O Devices

John Hyde

Intel University Press

The PC Platform Designers' Choice



intel

Copyrighted Material

Chapter 1
Adding I/O Devices to a Modern PC

The personal computer (PC) has been around for a long time, in computer years. The first IBM PC was announced in 1981, and since then we've all wanted to add hardware to our PCs. The power of the PC has grown—the number of tasks we want the PC to do has grown—and the number of devices we want to connect to the PC has grown. But until recently there has been a practical limit to the number of available ports one could connect devices to.

Enter the Universal Serial Bus, USB. You may have heard USB referred to as "the best thing" to happen to the personal computer for some time. The list of USB features is impressive:

- Hot-pluggable: I/O devices can be added while the PC is running.
- Ease of use: I/O device attachment is recognized by the PC and appropriate device drivers, and configuration is done automatically.
- Single connector type: all devices plug into the same socket type.
- High performance: 12 Mbps is much faster than existing serial and parallel ports.
- Up to 127 devices: there is no practical limit to I/O device expandability.
- Power supplied by cable: most devices will not need an additional power source.
- Power management: devices automatically power down when not in use.
- Error detection and recovery: errors are detected and transactions are retried to ensure that data is delivered reliably.
- External to the PC: there is no need to open the PC or design cards that must be installed in the PC.

Acknowledging the benefits of USB and designing an I/O device interface are two different things, however. Until now, you've had only the USB Specification to read for technical details. A specification by its nature usually provides few if any implementation examples. So, even after reading the specification, you still might not know how to design a simple I/O port for USB, let alone design a telephone or a camera-based I/O device.

Hands-On Examples

Most USB books, including the specification, position USB as a technical wonder, which it is. But these books lack practical, how-to examples that include schematics and code. That's where this book comes in. It is intended to help you add I/O devices to your PC, and I have tried to make this as much of a "cookbook" as I can. All the examples are fully documented, and prototype boards are available for you to build upon and expand your ideas and solutions. A variety of vendor solutions are demonstrated so you can choose the solution closest to your application.

I've used the term "PC" to refer to Intel-based computers running an operating system that supports USB (I used Windows 98 in the examples).

- The PC host software examples use Visual Basic to hide some complexities of operating system programming and allow the examples to be portable.
- The microcontroller examples are mainly in MCS51 assembler code; I chose this architecture because many manufacturers have used it as the basis for their intelligent USB controllers.
- The 8051 is an uncomplicated architecture with few programming "tricks," so the examples should be easy to port to another microcontroller family.

I have not repeated sections from the USB Specification but have included the full specification on the companion CD-ROM.

This book describes how to connect to a PC host the usual PC-type peripherals, such as scanners, proximity detectors, keypads, and printers, and also how to connect to everyday items such as lights, switches, motors, temperature sensors, speakers, video, and a telephone.

You will soon discover that adding I/O devices to a modern PC host is both easy and fun. The range of devices is limited only by your imagination.

How Much Technical Background Do You Need?

I assume you have some fundamental electronics and programming skills, but I don't expect these to be your major field. Instead, I assume that you want to **use** the PC to do something useful. This book takes a practical approach to adding devices to a PC. This task is much easier to do today when compared with previous generations of personal computers. Today we connect to an external, standard USB socket and don't even have to open up the case of the computer—a much more civilized approach and far less prone to error.

Focus of this Book

This book focuses on I/O device design. The book also covers PC host software to let us view and control our I/O devices. The hidden part of USB is the PC host software—we'll see that the implementation on Windows 98 uses a layered approach so that we can access USB features and services at a level suitable for our application. One of Intel's goals in working with Microsoft on their USB software was to eliminate the need for the user to write **any** OS-level software; almost all of the PC host software examples in this book use applications-level software.

When USB was being developed, a base assumption was the creation of "easy, low-cost I/O devices." The resulting standard is asymmetric with most of the complexity on the PC host side. The operating system and available host controller silicon take care of this complexity so we need deal only with the so-called "easy part," the I/O device. Hub design is presented, but to be honest, it's more productive for you to buy one of the many units available in the marketplace and instead focus your efforts on your unique value-added I/O subsystem.

The Modern PC: A Short History

What is a modern PC? And if you have an old PC, how do you make it modern? This section gives a brief history of the modern PC and introduces USB as a solution to simple, low-cost I/O expansion.

When IBM announced the PC, the company documented the PC internals thoroughly. The availability of this information, along with a strong desire to add hardware to the PC, caused many people to develop plug-in cards so the PC could monitor and even control their environment. Initially the slots inside the IBM PC didn't even have a name. The name ISA, for Industry Standard Architecture, was coined in the early 1980s. There were no official guidelines on how a PC should be expanded, so cards made by vendor A could not be used at the same time as cards made by vendor B. It took until the early 1990s for the ISA bus to be documented (*ISA & EISA Theory and Operation* by Edward Solari is the classic text for this information). Some developers took a lower-risk approach and created attachments that plugged into the serial and parallel ports. But this port resource was quickly depleted, and when more serial and parallel ports were added via plug-in cards, the internal design of the PC could not

support enough interrupts, DMA channels, or other system resources. Sometimes the interactions were subtle and failed only under certain conditions. The bandwidth of these data paths in and out of the PC was also quite low. A better solution was required.

Many PC and PC component vendors, including Compaq, Digital Equipment, NCR, IBM, and Intel Corporation, worked together to define a high-bandwidth expansion bus that was eventually called the Peripheral Components Interconnect bus, or PCI. This bus definition included configuration information and controls to alleviate the vendor-to-vendor conflicts of ISA. The PCI bus was included beside the ISA bus inside the PC host. Software support for automatic configuration was added to Windows so that PCI cards became easy to install. This was the start of Plug and Play, an industry-wide initiative that governs the way add-in hardware should identify itself.

PCI was an instant success for high-bandwidth devices but was seen by many to be excessive and complex for the simpler I/O devices. The PCI interface components typically cost more than a simple I/O device. Another main disadvantage was the location of the PCI connectors; just like their ISA predecessor, they were inside the PC case, which had to be opened, clamping screws undone, boards plugged exactly right into connectors that were difficult to identify, and then the system reassembled. This was discouraging for many would-be users because there was a perceived concern that the PC could be easily broken by these actions. PCI was a better solution but did not address "simple I/O."

Several PC-industry leaders worked together to define an external expansion bus. Driven by customer demand, it was essential that this bus be simple and low cost. Consumer focus groups demanded that PC expansion be as easy as connecting a VCR to a TV—you should not need to set switches or run software, and you must be allowed to plug and swap cables while the equipment is turned on.

Simple to Use and Low Cost

With the goals of simple-to-use and low cost to implement, industry leaders turned their attention to a high-speed serial bus that was later called Universal Serial Bus (USB). Serial was preferred over parallel because the cables would be cheaper, and it would be easier to implement dynamic configuration. "Dynamic configuration" means that the I/O subsystem can be extended or reconfigured by swapping cables while the PC is running. Rebooting a modern PC takes several minutes, so this situation has to be avoided in every solution.

A typical configuration includes one PC host and many I/O devices. To reduce costs, a master-slave implementation was chosen for USB. The PC host would be the master controlling all traffic on the serial bus—the additional silicon complexity needed to implement the controlling protocols would need to be implemented only once in the host. The slave I/O device could then be made simpler and therefore cheaper. This asymmetric solution means that the two ends of a cable are NOT equal—we need to know which end connects to the master and which end to the slave! The terminology adopted by the USB specification is "upstream" (toward the PC host) and "downstream" (toward the I/O device) (Figure 1-1). The upstream end of the cable controls the protocol and instructs the downstream end to reply at defined times.

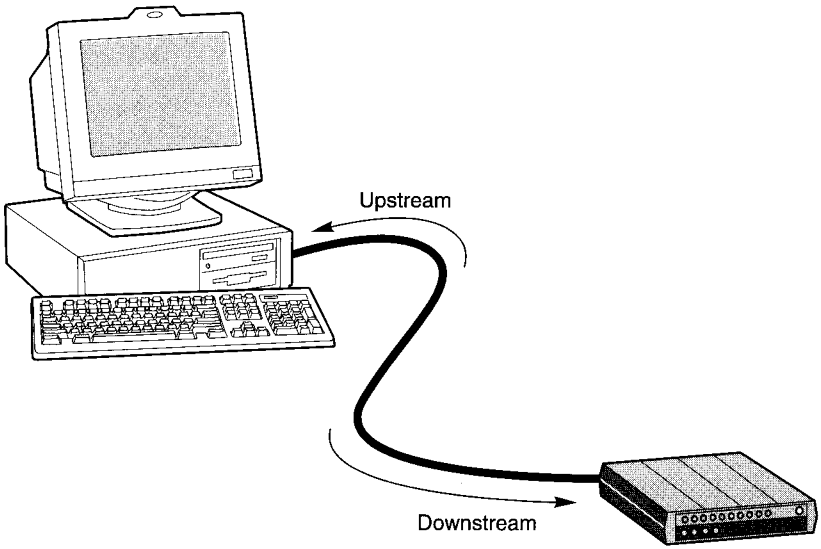


Figure 1-1. A USB cable has two different ends

Another major decision that would support both design goals was to officially supply power from the PC host. Different operating modes that specify power limits are defined, and a good understanding of this feature enables cheaper and easier-to-use I/O devices to be built. You can, for example, eliminate the cost of a power supply in simpler I/O devices and ease the design of more complex I/O devices. Some implementation trade-offs in the I/O devices could also be made when the system specifies minimum and maximum power levels.

USB TERMINOLOGY

The USB specification introduced new terms that are used throughout the USB literature. This section introduces those terms and presents an overview. Later chapters discuss each element in detail.

A typical configuration has a single PC host with multiple **devices** interconnected with USB **cables** (Figure 1-2). The PC host has an embedded **hub**, also called the root hub, which typically contains two USB ports.

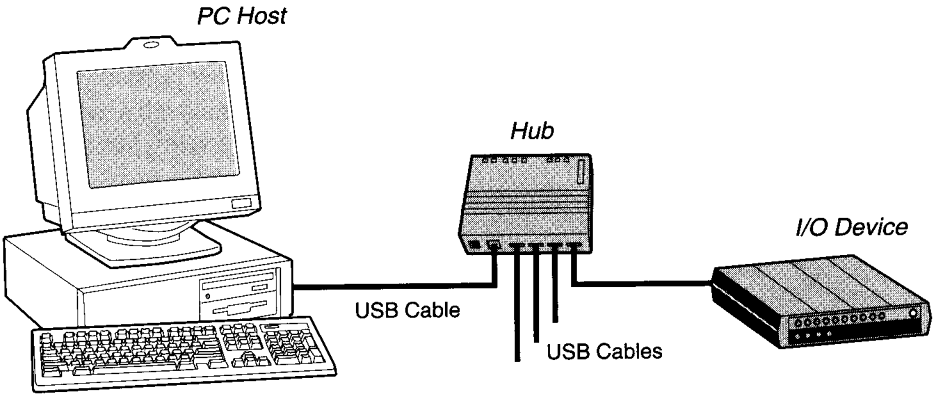


Figure 1-2. Standard USB terminology

Device configurations range from simple to complex:

- Hub: If a device contains only more USB ports, then it is called simply a hub.
- I/O device: An I/O device adds a capability to the PC host. It has a single upstream connection and interacts with the real world to create or consume data on behalf of the PC host.
- Compound device: If a device includes some I/O functionality as well as hub functionality, it is called a compound device.
- Composite device: If a single device implements two or more sets of diverse functions, it is called a composite device (for example, a keyboard with embedded speakers).

As far as the PC host is concerned, devices are the important feature, and up to 126 can be interconnected using external hubs up to five levels deep (Figure 1-3).

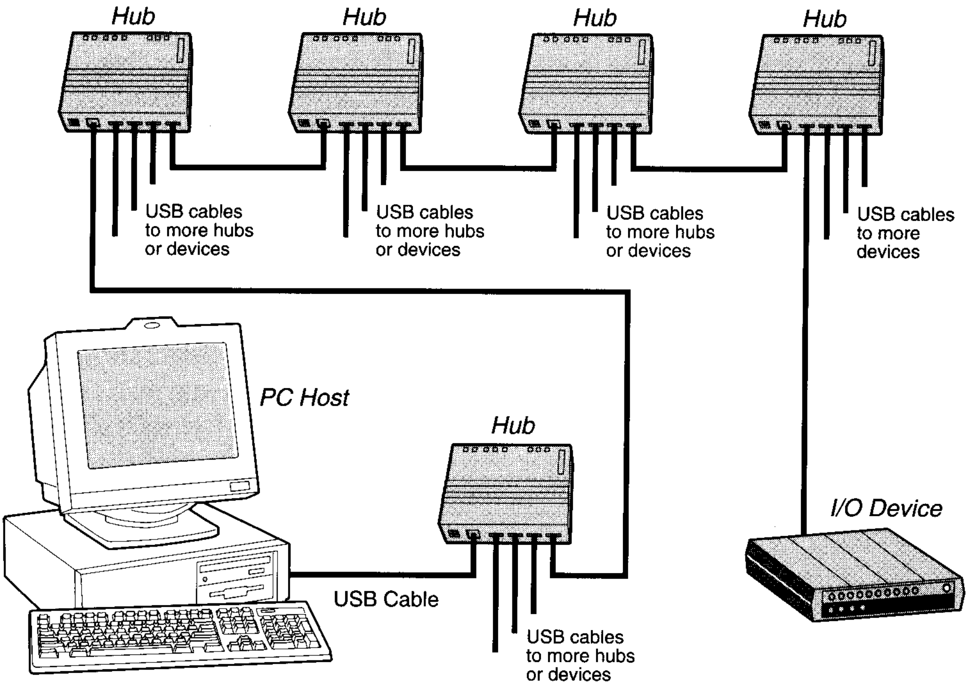


Figure 1-3. Devices can be nested five hub levels deep

PC Host

A typical configuration has a single PC host. You can connect multiple PC hosts using USB, and this special case is discussed in Chapter 6. The PC host runs USB-aware operating system software that supports two distinct functions—initialization and run time.

The USB initialization software is active at all times and not just during PC host power-on. Because the initialization software is always active, USB devices can be added and removed at any time. Once a device is added to a PC host, the device is enumerated by the USB initialization software and assigned a unique identifier that is used during run-time. This **enumeration** process is described in detail in Chapter 3.

Figure 1-4 shows how the USB host software is layered; layering supports many classes of devices. A **class** is a grouping of devices, with similar characteristics, that can be controlled by a generic class device driver. Examples of classes include mass-storage devices, communications devices, audio devices, and human interface devices. A single I/O device can belong to multiple classes. If a device fits neatly into one or more of these predefined classes, then you don't need to write operating system software, such as a device driver. The software structure allows you to focus more of your I/O device design efforts on the function and usability of the I/O device and less on the inner workings of an operating system. Vendor-defined commands are also available for those who want specific functionality for their device only.

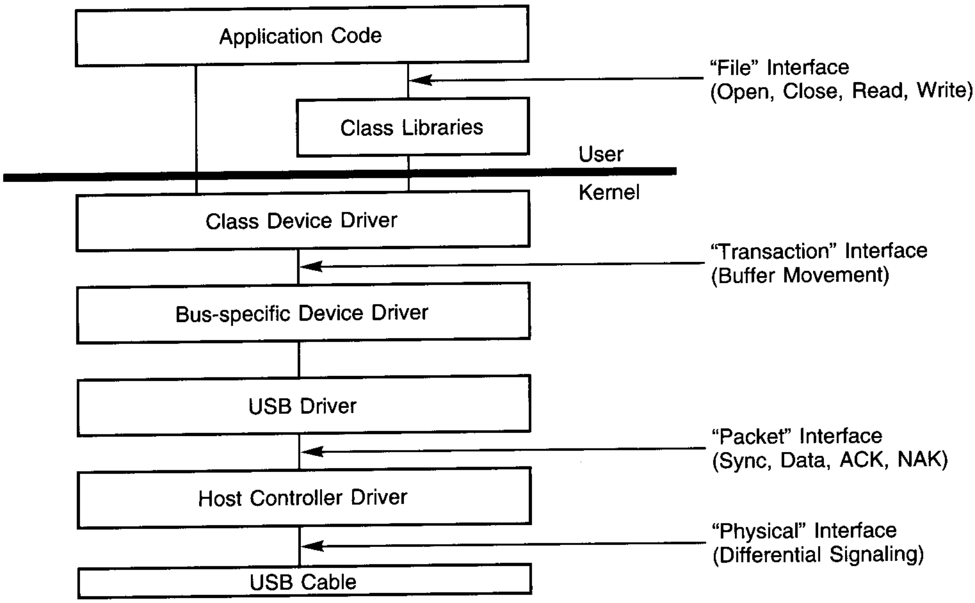


Figure 1-4. PC host software for USB is defined in layers

USB Cable

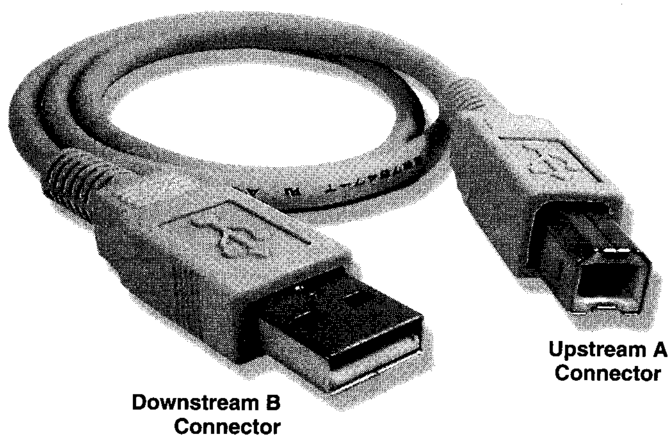
A USB cable includes both power supply and data signals.

Power supplied by the USB cable is an important benefit of the USB specification. A simpler I/O device can rely on the USB cable for all its power needs and will not require the traditional "black brick" plugged into the wall. The power resource is carefully managed by USB with the hub device playing the major role.

A hub or I/O device can be self-powered or bus-powered.

- Self-powered is the traditional approach in which the hub or I/O device has an additional power cable attached to it. If the hub is self-powered, it can make up to 500 mA available for each of its downstream ports.
- A bus-powered device relies solely on the USB cable for its power needs. If the hub is bus-powered, then it has a maximum of 500 mA available. The hub will use 100 mA for itself and have only 100 mA available for each of four downstream USB ports (unless the socket is suspended).

The USB cable connectors were specially designed with the power pins longer than the signal pins so that power is always applied before signals. Two different connector types were defined to ensure that illegal configurations could not be made (Figure 1-5). An A-type connector defines the upstream end of the cable, and a B-type connector defines the downstream end of the cable.



Courtesy of Newnex Technology Corp.

Figure 1-5. Different connectors define the ends of a USB cable

The maximum length of the USB cable was an engineering trade-off. Electrical signals on the wires take a finite time to travel the length of the cable. The cables must be series-terminated at the characteristic impedance of the cable to minimize signal reflections so we do not have to wait for the signals to "settle." The shorter the cable, the higher the signaling rate can be. A data rate of about 1 megabyte per second (MBps; implemented as 12 megabits per sec, Mbps) was chosen to support today's current desktop peripherals and many of tomorrow's devices. This results in a maximum cable length of 5 meters, which is sufficient for most desktop applications.

A cable with a 12-Mbps data rate makes a good antenna, unfortunately, so cable shielding is required. But shielding adds to the cost of the cable. An alternative approach is to reduce the signaling rate to 1.5 Mbps so that shielding is not required; this approach is cheaper. The USB specification allows a system to use a mixture of 12-Mbps and 1.5-Mbps devices.

The USB specification defines differential signaling on its data wires (Figure 1-6) to reduce the effects of induced system noise. Chapter 2 explains this in detail.

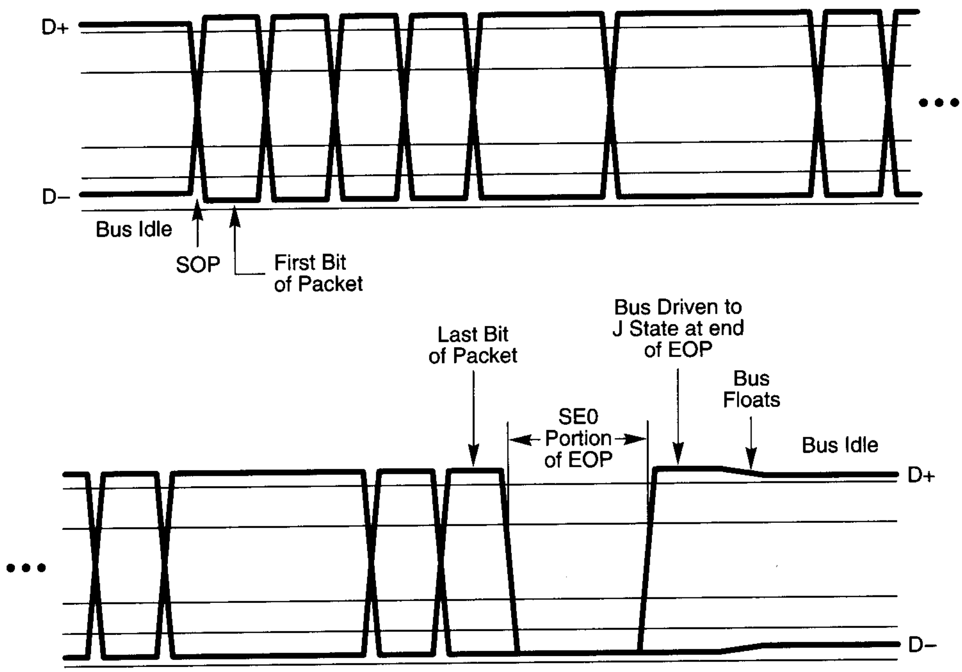


Figure 1-6. USB defines differential signaling

The USB bus is a point-to-point connection that operates in half-duplex mode. The fundamental element of communication on the USB bus is a **packet,** and defined sequences of packets are used to build a robust communications channel. Chapter 2 describes packet types and communications protocol in detail.

Hub Device

The hub has two major roles: power management and signal distribution.

An external hub has one upstream connection and multiple downstream connections. The USB specification does not limit the number of downstream connections, but seven seems to be the practical limit. The most popular hub size is four.

A hub can be self-powered or bus-powered.

- Self-powered is the traditional approach in which the hub or I/O device has an additional power cable attached to it. If the hub is self-powered, it can make up to 500 mA available for each of its downstream ports.
- A bus-powered device relies solely on the USB cable for its power needs. If the hub is bus-powered, then it has a maximum of 500 mA available. The hub will use 100 mA for itself and have only 100 mA available for each of four downstream USB ports (unless the socket is suspended).

Because of the power limitation, I do not recommend bus-powered hubs except in exceptional situations. A bus-powered hub can support only 100 mA on each of its downstream USB ports. Although this is enough to enumerate all I/O devices, it is typically not enough for most I/O devices to operate. The current implementation of Windows 98 does not flash lights or make warning sounds if an enumerated device can't operate because of lack of hub power, so the user is left wondering why the newly attached device doesn't work—not a good user experience. This situation can get worse: Nothing prevents someone from adding a second bus-powered hub onto a port of the first bus-powered hub; the second hub uses all of the 100 mA for itself and cannot supply enough power to even enumerate additional I/O devices. This confuses a user even more. A future release of Windows 98, and Windows 2000, will alert the user if a high-power device is attached to a low-power hub and will recommend a better system configuration.

When first attached to a USB socket, an I/O device (or hub) can always expect 100 mA to be available. The device uses this power to operate during the enumeration stage. The device may NOT use more than 100 mA until it is configured; if it does, the power source on the USB socket will be removed and an error sent to the PC host. If the I/O device requires more than 100 mA for run-time operation, it can request up to 500 mA from the hub. If the hub can supply this power, it does so; otherwise, the I/O device is not configured, and an error message is sent to the PC host. If the I/O device requires more than

500 mA for run-time operation, the device must be self-powered and will need a power cord.

It is worth the design effort to reduce your I/O device power to less than 500 mA because this eliminates the need for an external power source.

An I/O device is required to power itself down, or suspend, when there is no activity on the USB bus. From the USB specification definition, "no activity" means no bus signaling for 3 ms. The maximum amount of power allowed to be drawn during a suspend is 0.5 mA; this is a larger design challenge, as we shall see in later chapters.

I/O Device

An I/O device creates data for, or consumes data from, the real world, as shown in Figure 1-7. A scanner is a good example of a data creator, and a printer is a good example of a data consumer.

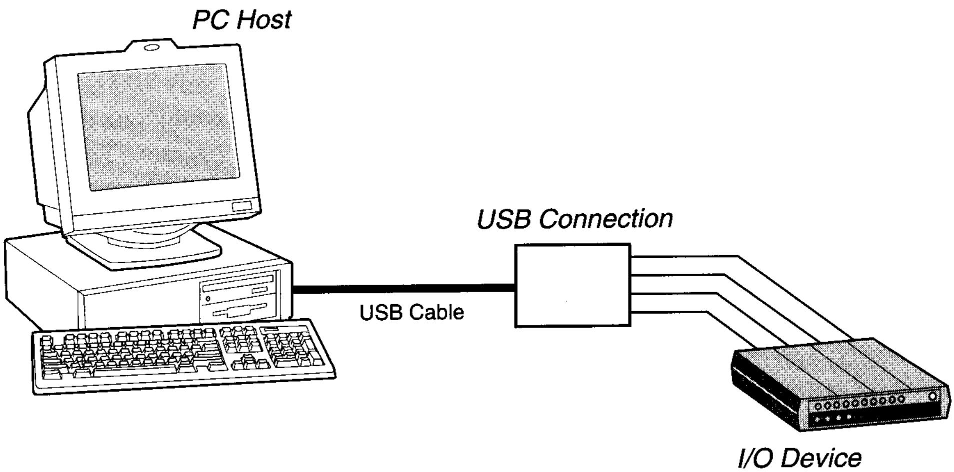


Figure 1-7. I/O device connects USB to the real world

A single I/O device could support both a data creator and a data consumer, so a different software driver may be required for these two distinct tasks. The software, or logical, view of the USB connection is shown in Figure 1-8. The logical view is deliberately general in nature so that all types of real-world connections can be made. This diagram is best explained from the bottom up.

The term **endpoint** is used to describe where data enters or leaves a USB system. An IN endpoint is a data creator, and an OUT endpoint is a data consumer.

A typical real-world connection may need multiple IN and/or OUT endpoints to implement a reliable data delivery scheme. This collection of endpoints is called an **interface** and is directly related to a real-world connection. The operating system will have a software driver that corresponds to each interface.

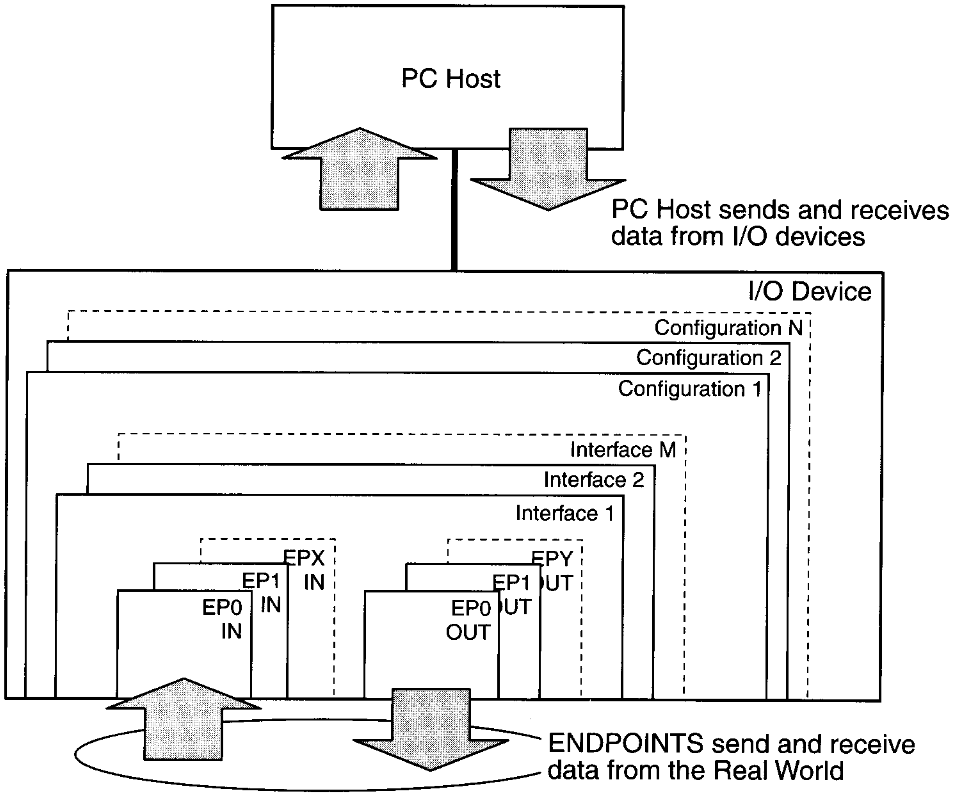


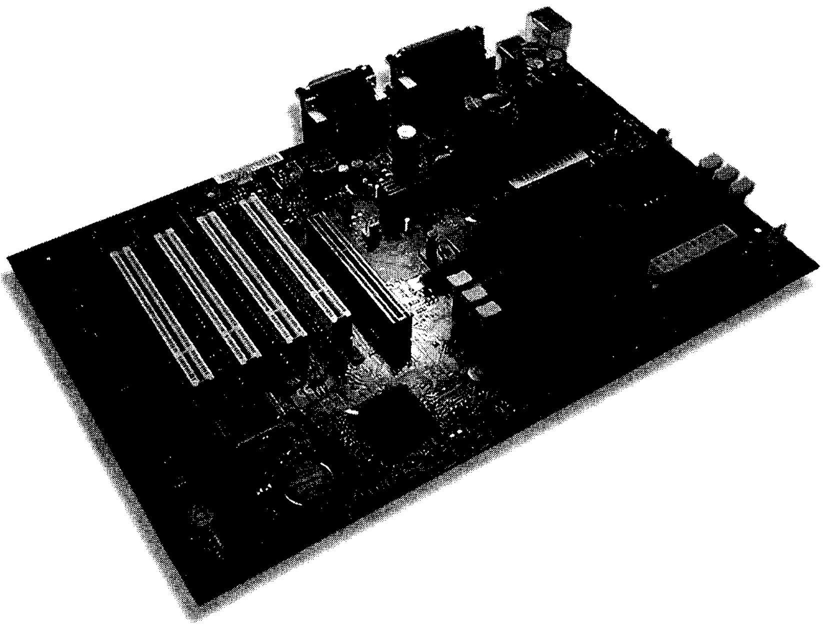
Figure 1-8. Logical view of an I/O device

Some real-world devices may have multiple interfaces—a telephone, for example, has a keypad interface and an audio interface. The operating system will manage the keypad and audio using two separate device drivers. This decomposition of complex devices into smaller logical interfaces means that building-block software elements can be quickly used to manage these complex devices. We don't have to invest the time and effort to write a special telephone device driver—we can be operational with the class drivers already included in the operating system. All the interfaces run concurrently.

A collection of interfaces is called a **configuration,** and only one configuration can be active at a time. A configuration defines the attributes and features of a specific model. Using configurations allows a single USB connection to serve many different roles, and the modularity of this system solution saves development time and support costs.

IMPACT OF USB ON PC HOST

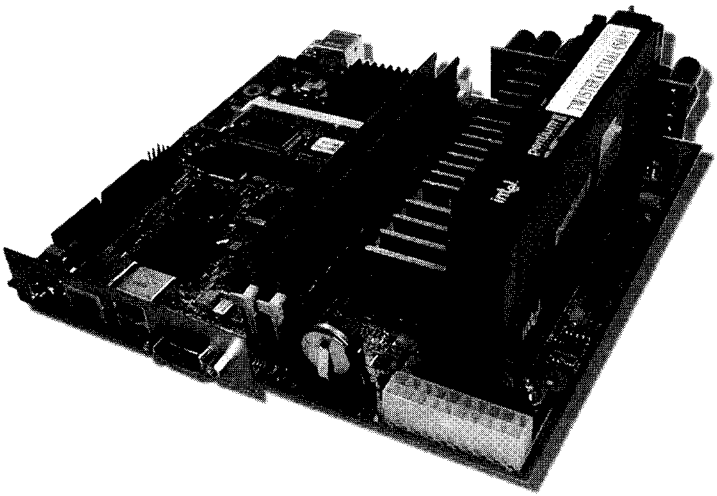
As well as simplifying I/O device design, USB can have a major impact on the PC host design. All of the basic features of a PC are implemented in highly integrated components, and a view inside today's PC shows a motherboard similar to Figure 1-9; most of the physical space is taken up by PCI and/or ISA slots!



Courtesy of Systems Group, Intel Corp.

Figure 1-9. Typical PC host motherboard

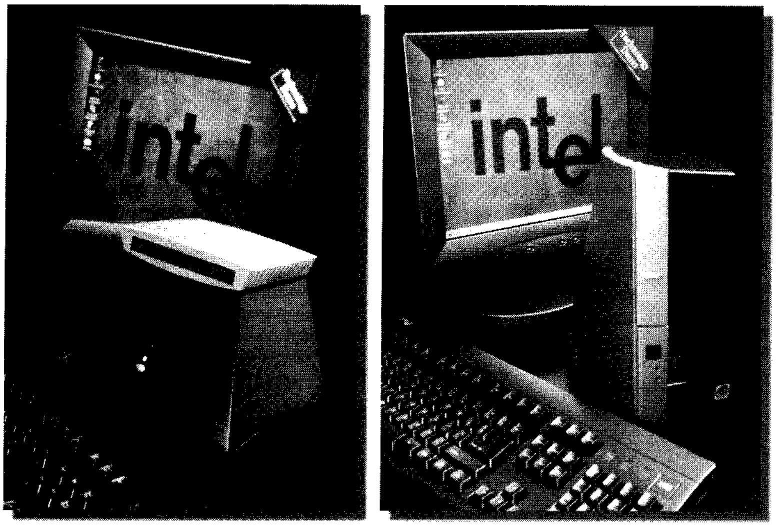
If all I/O expansion could be external, then there would be no requirement for expansion slots on the motherboard. As well as saving physical space, we also have savings in the power supply design. ISA slots have a power allowance of 25 watts, and PCI slots have a power allowance of 10 watts. This means a PC with no I/O expansion slots ("slotless") needs to have only a 50-to-80-watt power supply rather than a 180- or 240-watt unit. Less heat will be generated inside the system, so a single cooling fan will be adequate. Figure 1-10 shows an example of a slotless motherboard that is only 8 inches square.



Courtesy of Desktop Architecture Laboratory, Intel Corp.

Figure 1-10. Slotless motherboard (8 inches square)

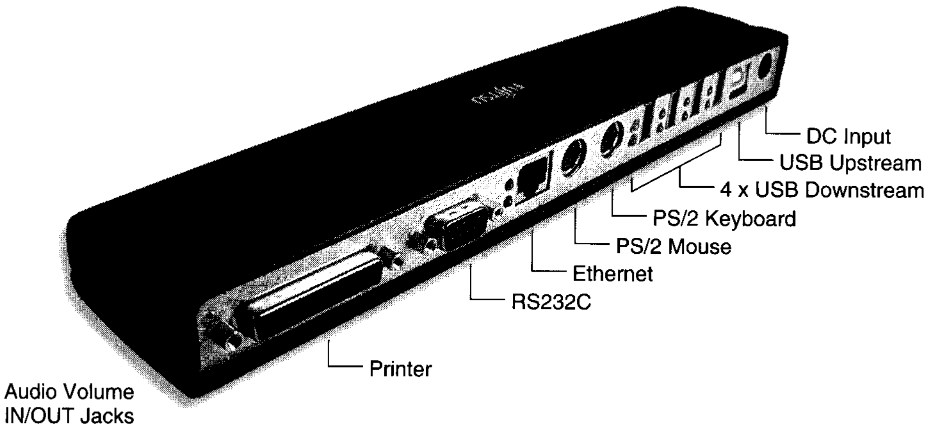
Figure 1-11 shows two concept platforms built around a slotless motherboard. The platforms contain the same motherboard and the same peripherals, a 6-GB hard drive, and a DVD-ROM, and they differ only in the physical placement of the subunits. These PCs are very quiet and don't look like typical "beige-box" PCs. They will be attractive for several new applications such as home use.



Courtesy of Desktop Architecture Laboratory, Intel Corp.

Figure 1-11. Concept PCs built with slotless motherboard

These concept platforms also introduce new I/O paradigms. There are many peripherals available today that attach to the serial or parallel ports of the traditional PC. A peripheral device expander (Figure 1-12) could be used to attach these peripherals to a concept platform. Software drivers on the PC host would redirect I/O requests to this expander, so no changes in applications software would be required. The device shown in Figure 1-12 is from Fujitsu Ltd.; it additionally includes a four-port hub, an Ethernet connection, and sound. Although targeted at USB-enabled laptop computers, the device expander would complement a slotless desktop implementation.

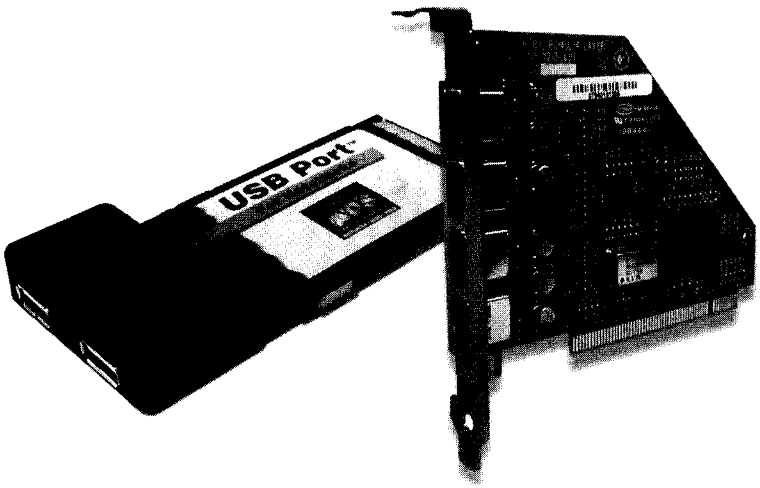


Courtesy of Fujitsu Limited.

Figure 1-12. Integrated hub and I/O expander

CHAPTER SUMMARY

The Universal Serial Bus (USB) was the result of a tremendous amount of cooperative industry effort, and its inclusion and operating system support in a PC defines a modern PC. If your old notebook or desktop PC does not have USB sockets, these can be added via a PCMCIA card or a PCI card (see examples in Figure 1-13).



Courtesy of ADS Technologies, Inc. (left) and Entrega Technologies, Inc. (right).

Figure 1-13. Adding USB capability to a PC host

The PC host software is layered and interfaces are defined to allow the simple addition of application programs. The higher up the USB software stack we go, the farther we are from the actual I/O devices we are controlling. This abstraction allows the software and hardware to be developed on different schedules by different people. The hardware and software communicate via standardized interfaces that have the added benefit that they can be either swapped-out or upgraded independently. This new freedom will promote more diverse uses of the PC platform.

CHAPTER 2
CLOSE TO THE WIRE

This chapter studies host PC requests on the USB bus from the bottom up:

- We look at the signaling on the wires and learn that it reveals a packetized structure.
- We learn that sequences of packets are used to generate transactions.
- We learn that three types of transactions are used during run time (interrupt, bulk, and isochronous), and control transactions are used during enumeration and run time.
- We list the available control requests that a PC host can use to interact with an I/O device and a hub device.

DIFFERENTIAL SIGNALING

If you were to put an oscilloscope on the USB data wires, you would see a pair of differential signals (Figure 2-1). This section assumes we are observing a full-speed connection. (A low-speed connection is a little different; see "Differences for a Low-speed Device" at the end of this chapter.) All communication is initiated from the root hub (see Chapter 3 for one exception) and is repeated by the external hubs. The USB data wires are connected point-to-point and the signaling is half-duplex, which means that only one end of the wire is driven at a time. The protocol, presented below, expects the two ends of the wire to take turns transmitting information. In the following discussion, the upstream device is the **transmitter,** and the downstream device is the **receiver** unless noted.

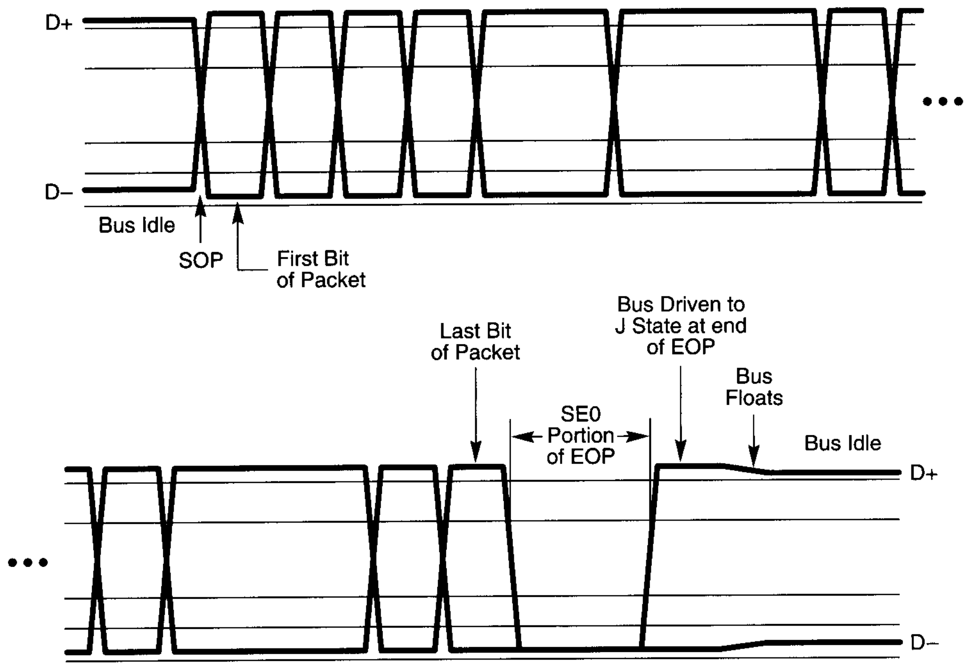


Figure 2-1. USB data wires generally driven differentially

The USB data wires do not include a CLOCK signal, so the communication between nodes is described as asynchronous. All devices on the bus communicate at a nominal 12 MHz, but each device has its own 12-MHz oscillator. When receiving a signal from the bus, a device will typically oversample the signals (typically at 48 MHz) so that transitions can be better detected. Later we'll see that a CLOCK signal is embedded in the data.

An IDLE full-speed bus has D+ high and D– low. Some call this the "1" state of the bus, but to avoid later confusion and provide consistency with the USB specification, we'll call this the "J" state.

THE FUNDAMENTAL PACKET

The fundamental element of communication on the USB data bus is a **packet**. A packet consists of three pieces: a start, some information, and an end, as shown in Figure 2-2.

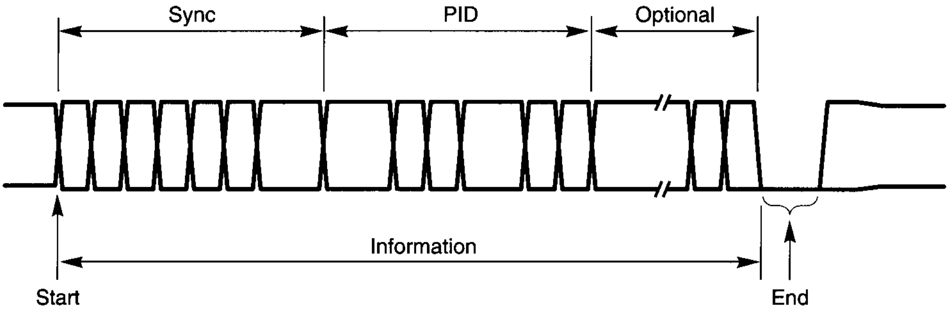


Figure 2-2. Anatomy of a packet

The start of a packet is signaled by a transition out of the J state into the K state; both D+ and D– switch to the opposite polarity. Six more transitions are driven by the transmitter on the next available 12-MHz bit times, to produce a SYNC sequence. The receiver uses this SYNC sequence to tune its receive clock with the transitions of the received data, thus ensuring reliable reception of the information portion of the packet. The SYNC sequence is eight bits of data (KJKJKJKK), and the packet information starts in bit-time eight.

The packet information varies from one byte to 1025 bytes. The first byte is always a Packet Identifier, or PID, that will define how the other information bytes should be interpreted. A packet identifier byte is formed with 4 bits and the complement of these 4 bits; this redundancy allows the receiver to error-check the PID. Of the possible 16 PID types that this encoding scheme allows, ten are currently defined and six are reserved as shown in Table 2-1.

Table 2-1. Ten packet types are defined

PID Value	Packet Type	Packet Category
0101	SOF	Token
1101	SETUP	Token
1001	IN	Token
0001	OUT	Token
0011	DATA0	Data
1011	DATA1	Data
0010	ACK	Handshake
1010	NAK	Handshake
1110	STALL	Handshake
1100	PRE	Special
Others	RESERVED	RESERVED

There are four categories of packets. **Token** packets are used to set up **Data** packets, which are acknowledged by **Handshake** packets. There is also one **Special** packet used to implement a low-speed connection. The next section discusses each packet type in detail. Each will be presented as a discrete building block, and we'll use a sequence of these building blocks to define a robust communications channel.

The last part of a packet is an End-Of-Packet identifier. Both D+ and D– are driven low for two bit times to signal this End-Of-Packet. This is not a differential signal. It is an easily identified single-ended zero, or SE0.

As an aside, it is worth mentioning the term "bit stuffing" that is used in the USB specification. The data lines on the bus reflect the information that is being transmitted. There is no separate CLOCK signal, and the receiver relies on regular signal transitions to maintain synchronism with the transmitter. A nonreturn to zero inverted, or NRZI, transmission encoding is used on the data lines. The encoding and decoding occur at the transmitter and receiver, respectively, so the process is transparent to all other parts of USB. The process will, however, be observed on the bus.

The NRZI protocol requires the following:

- Toggle each bit time for multiple data 0s.
- Do not toggle each bit time for multiple data 1s.
- Toggle each data 1 to 0 pair.
- Do not toggle each data 0 to 1 pair.

This scheme results in no bus transitions for long sequences of data 1s, and as a result, the receiver could lose synchronism. The USB specification requires that a 0 data bit is added, or stuffed, after six consecutive 1 data bits to ensure that the bus transitions often enough. We do not have to account for this during any part of the design, but if you put an oscilloscope on the bus, you will see these "extra" bits.

Different Packet Types

Start-Of-Frame Token Packet

The root hub transmits a SOF packet every 1.0 milliseconds. The time between two SOF packets is called a **frame**. Within this frame, there are (1.00 ms/12 MHz) = 12,000 theoretical bit times or 1500 bytes of data, maximum. The SYNC and EOP overhead will reduce this number to a practical upper limit of 1200 bytes of data that could be transmitted in each frame.

The USB signaling scheme includes robust error checking. This Cyclic Redundancy Check, or CRC, code is generated by the transmitter and received by the receiver. The receiver also generates a CRC code over the same data bits and compares this with the received code. If they are different, the packet is rejected and not acted upon.

A SOF packet has 11 bits of data and 5 bits of CRC error checking (Figure 2-3).

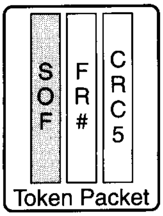


Figure 2-3. Start-Of-Frame packet

The 11 bits of data in a packet is a monotonically increasing frame number. It has no absolute value but is used by real-time devices to synchronize their data transfer. This number rolls over every 2048 milliseconds, or approximately every 2 seconds.

So with no other USB activity on the bus, our oscilloscope would see a SOF packet every millisecond. Any device attached to the bus could receive this SOF packet and use it as a regular "heartbeat" signal. We shall see in Chapters 11 and 12 that real-time devices, such as audio and video, rely heavily on this regular SOF packet.

The SOF packet is the only packet that does not have a destination address. It is broadcast for all USB devices to use and does not require an acknowledgment.

Setup, IN, and OUT Token Packets

The other three token packets, SETUP, IN, and OUT, have the same format as shown in Figure 2-4. They contain a 7-bit device address, a 4-bit endpoint address, and a 5-bit CRC.

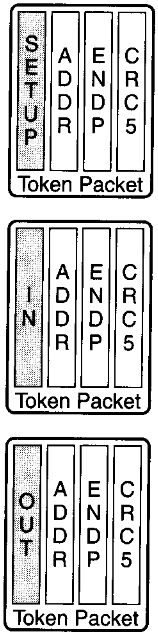


Figure 2-4. SETUP, IN, and OUT token packets

These three token packets are used to set up data transfers between the root hub and a specific data source or sink on a specific device (this data source or sink is called an **endpoint**; it is defined in the next chapter).

- An IN packet sets up a data transfer from the device to the root hub, and an OUT packet sets up a data transfer from the root to the device.
- IN and OUT packets can address any endpoint on any device.
- A SETUP packet is a special case of an OUT packet; it is "high priority," which means that the device MUST accept it even if this means aborting a previous command, and it is always targeted at the bidirectional control endpoint 0.

Data Transfer Packets

The data transfers initiated by the SETUP, IN, and OUT token packets are implemented with DATA0 and DATA1 packets (Figure 2-5).

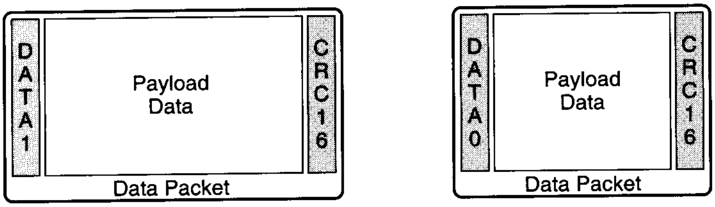


Figure 2-5. Data transfer packets

A data transfer packet can have a payload varying from 0 to 1023 bytes and a 16-bit CRC. (Some restrictions apply to payload size; these are detailed in the sections about the different transfer types further on in this chapter.) There are two data packet types to provide error detection. A transmitter will alternate DATA0 and DATA1 packets, and the receiver should check that alternate DATA0 and DATA1 packets are being received. For example, if two DATA0 packets are received back-to-back, this indicates a loss of a DATA1 packet and is an error condition.

Handshake Packets

Handshake packets are used by a receiver to indicate the good, bad, or ugly reception of token and/or data packets. Figure 2-6 shows the three handshake packet types: ACK, NAK, and STALL.

A handshake packet consists only of PID. There is no CRC; the redundant encoding in the PID provides the required error-checking.

- An ACK handshake indicates the successful reception of a token and/or data packet.
- A NAK handshake indicates that the receiver is currently too busy or doesn't have the resources to deal with the token and/or data packet right now. A device is allowed to NAK all transactions, except a SETUP token, while the root hub is not allowed to NAK any transactions. The root hub is inside a PC, so it should always have the time and resources to receive a packet; this will simplify the buffering on an I/O device.

- If something is very wrong at the device, then a STALL handshake is used to tell the root hub that some help is required. For example, an I/O device will generate a STALL handshake for all commands that it doesn't understand.

There also exists a special PRE packet; for a discussion of this packet type, see "Differences for a Low-speed Device" at the end of this chapter.

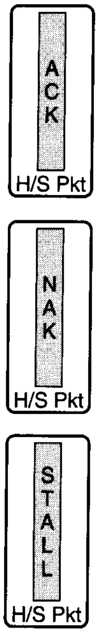


Figure 2-6. Handshake packets

BUILDING A TRANSACTION

A predefined sequence of packets is used to define data movement between the root hub and an I/O device. Each transaction **must** occur within the same frame; a transaction is not allowed to straddle multiple frames. Once a sequence is started, it must be completed with no intervening packets from other transactions.

The USB specification defines four different transaction types to handle different types of data that will be transmitted over the bus. All the transactions use the same building-block packets just presented but differ in how the packets are scheduled and the response to an error. The PC host software has two parameters to deal with—delivery **time** accuracy and delivery **quality** accuracy. The time and quality attributes have different importance to different data types. Table 2-2 summarizes the four transaction types and highlights the delivery attributes of each type.

Table 2-2. Four transaction types

Type	Important Attributes	Maximum Size	Example
Interrupt	Quality	64 (8 for LS)	Mouse, keyboard
Bulk	Quality	64	Printer, scanner
Isochronous	Time	1024	Speakers, video
Control	Time and Quality	64 (8 for LS)	System control

The PC host guarantees time accuracy by reserving portions of a frame for isochronous and control transfers.

The PC host guarantees quality accuracy by using a handshake mechanism. In general:

- If data is received correctly, then an ACK handshake is generated.
- If there is a problem with the data transfer, a NAK handshake is generated.
- If the data receiver is confused, a STALL handshake is generated.

The next section looks at each transfer type individually and studies their characteristics.

In the following diagrams about the different types of transfers, packets transmitted by the root hub are shown in white and packets transmitted by the I/O device in black. Only full-speed transactions are described here; low-speed differences are covered at the end this chapter.

Interrupt Transfers

Using the term "Interrupt Transfer" for this transaction is very generous. Hardware designers think of an Interrupt as an immediate response to an external event. But this is not how USB Interrupt Transfers occur. Instead, the root hub **polls** the I/O device to inquire if it needs attention. The root hub can poll as often as every frame, which is 1,000 times a second, but for many mechanical or human-related devices, polling once every 10 frames, or 100 times a second, will be an ample response time. If your I/O device needs attention faster than 1 millisecond, then the microcontroller should preprocess the data and handle a response locally, and a status update should be sent to the PC host at a convenient time later.

Figure 2-7 shows several interrupt transfers.

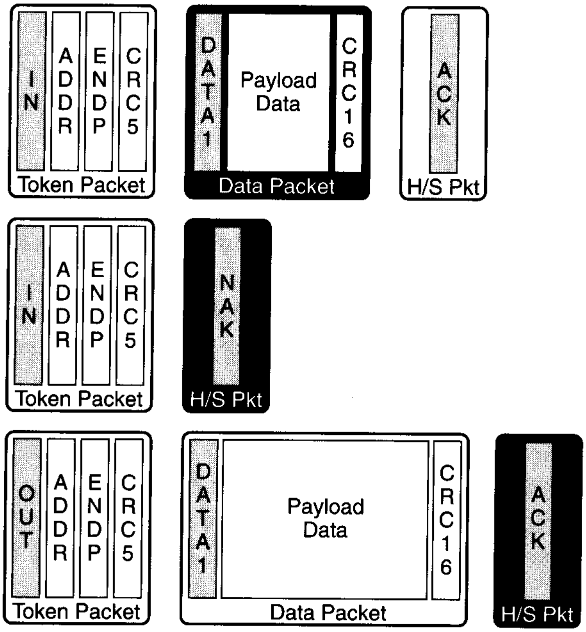


Figure 2-7. Interrupt transfers

The first sequence in Figure 2-7 shows a successful data transfer from the I/O device to the root hub. The data payload size varies from 0 to 64 bytes (0 to 8 for a low-speed device). The ACK is a positive response for a valid data transfer. To be able to respond so quickly, the I/O device must have had the data ready and be waiting for an IN token. If the I/O device did not have data ready, it would generate a NAK as shown in Figure 2-7 (center), and the root hub would retry in the next frame.

The third sequence shows a successful data transfer from the root hub to the I/O device. Note that OUT transactions were added in USB Specification V1.1. In this example, the I/O device must have had a buffer ready to accept the data, because it was able to respond immediately.

If the I/O device has no new data since the previous time it was polled, it can respond with a NAK. This preserves bus bandwidth, and the transaction will not be retried until the next scheduled polling interval.

If the I/O device is confused, it responds with a STALL, and the root hub alerts higher levels of software so that the situation at the I/O device can be resolved.

Bulk Transfers

The packets used to implement bulk data transfers are identical to those used for interrupt transfers—the differences are in the allowable data sizes and the system scheduling and response to a NAK handshake. USB provides "good delivery" of bulk data. The transfer quality is guaranteed such that no data is lost, but the transfer time is not guaranteed. Bulk transfers use the time available in a frame after all of the guaranteed transfers have been scheduled. This means that a long printout to a USB printer will take longer on a busy USB bus. On the other hand, if there is a lot of time available within a frame for bulk transfers, the PC host can schedule multiple transfers to the same I/O device in a single frame. The data packets will alternate between DATA0 and DATA1 tokens for error checking.

An I/O device can NAK a data packet if, for example, the device has no buffer space to put the incoming data in. The PC host will retry the transfer in the next frame.

Isochronous Transfers

The packets used to implement isochronous data transfers are similar to those used for bulk transfers but have different system scheduling and are not acknowledged. Before a PC host agrees to support isochronous data transfers to/from an I/O device, the PC host negotiates a guaranteed data delivery schedule. Isochronous transfers occur **every** frame, and the PC host will ensure there is available bandwidth within the frame before agreeing to set up the connection. Once set up, the I/O device is guaranteed a slice of every frame; however, the position within the frame is not guaranteed, so the I/O device must buffer the data and allow for delivery time jitter.

Figure 2-8 shows the isochronous packets; there is no handshake packet with each data transfer. Isochronous packets are time-dependent, and the late delivery of a packet is as useless as no delivery, so bad packets are not retried. An I/O device will typically use the data from a prior packet again. DATA0 tokens are used with a data payload that varies from 0 to 1023 bytes.

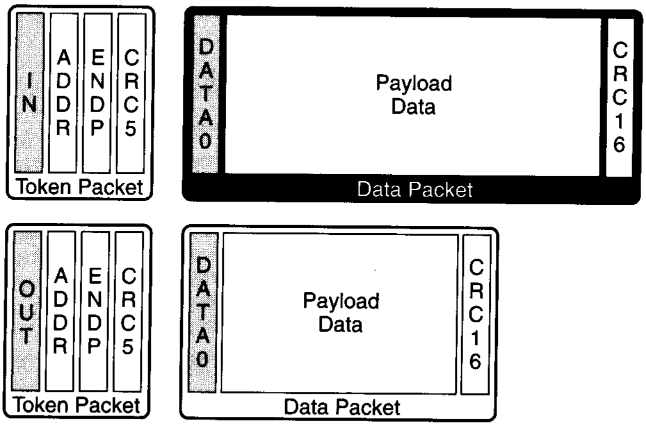


Figure 2-8. Isochronous packets

Control Transfers

Control transfers are the most complicated. Control transfers need a lot of system protocol overhead to ensure that the commands and data are correctly received. This is not a major concern for system performance, because these transfers typically occur only during enumeration or command initialization. A control transfer is divided into three phases, each of which uses some of the building-block packets already defined. These phases, shown in Figure 2-9, always start with a setup phase and end with a status phase. The data transfer phase is optional. All control transfers address endpoint 0 on the targeted I/O device.

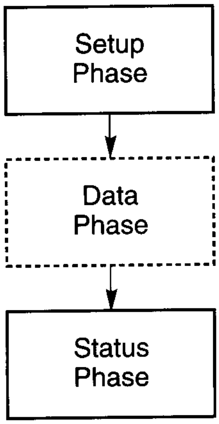


Figure 2-9. Control transfer phases

The setup phase consists of a setup packet, a DATA0 packet, and a handshake packet (Figure 2-10). The DATA0 packet always contain 8 bytes, and the format of this data is predefined (and presented in the next section). The handshake packet is always an ACK, because the I/O device is not allowed to NAK or STALL a setup packet. A setup packet must always be accepted even if this requires aborting the execution of a previous control transfer request.

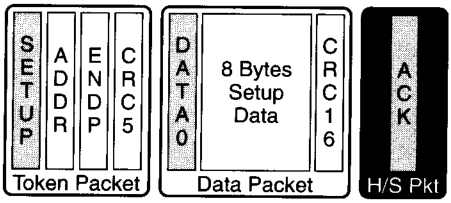


Figure 2-10. Setup phase of a control transfer

The setup phase will specify if a Data Phase is required. Some setup commands can be completely specified by the 8 bytes of data in the setup phase, and others require more data to be written to, or read from, the I/O device. Figure 2-11 shows a typical control read data from the I/O device.

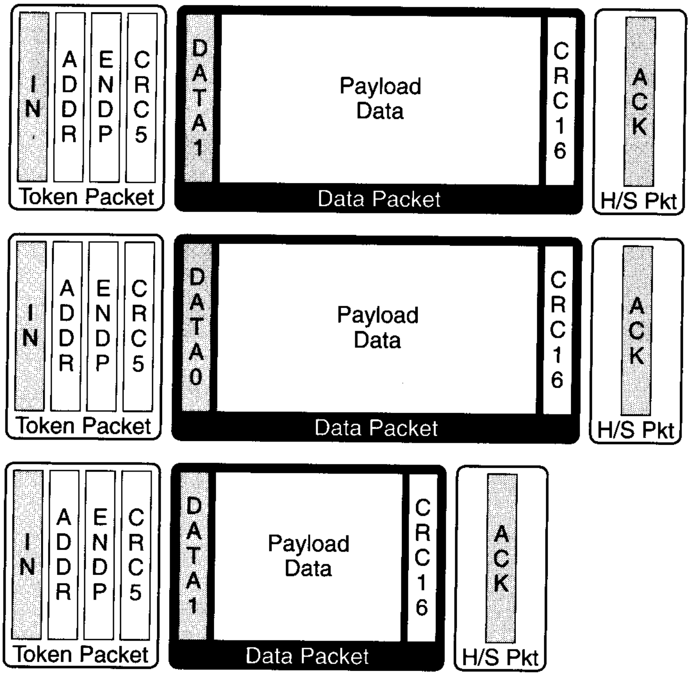


Figure 2-11. Control read data phase

Some setup commands require a lot of data to be read from the I/O device. If there is more data than can be contained in a single packet, then the I/O device must send the data in multiple packets; Figure 2-11 shows a multipacket response. The maximum size of a packet is implementation-dependent but must be a minimum of 8 bytes. Other legal sizes are 16, 32, and 64. The next chapter discusses how a particular I/O device informs the root hub of the maximum-sized packets that will used.

The root hub may want to provide more data to the I/O device—this would be specified in the setup phase. A control write data phase is used in this case. The example in Figure 2-12 uses multiple packets to supply all of the data.

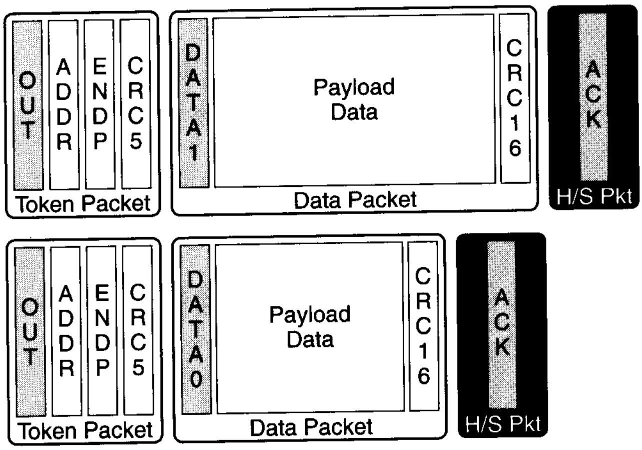


Figure 2-12. Control write data phase

A control transfer always ends with a status phase (Figure 2-13):

- If there is no data phase, then the I/O device is acknowledging receipt of the setup phase.
- If the data phase is a control read, then the root hub is acknowledging receipt of all the data from the I/O device.
- If the data phase is a control write, then the I/O device is acknowledging receipt of all the data.

The status phase is an IN packet if the I/O device is providing the status or an OUT packet if the root hub is providing the status. A zero-length data packet is used to signify success, and a NAK or STALL response denotes an error condition.

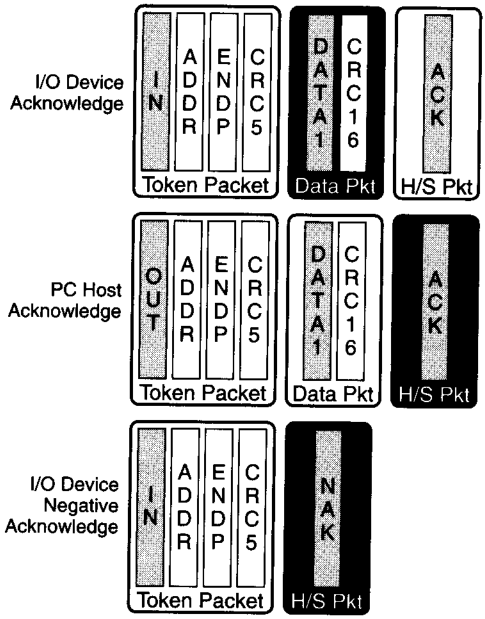
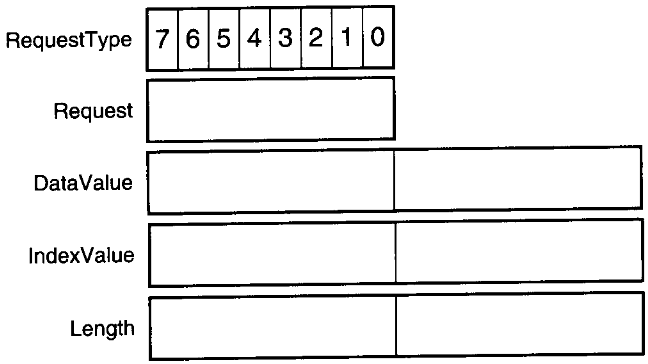


Figure 2-13. Final status phase of a control transfer

PC Host Requests

Figure 2-14 shows a fixed format for the 8-byte DATA0 packet within a control transfer request.



Definition of Request Type Bit Fields

Bit 7	0 = Transfer Host Data to Device		1 = Transfer Device Data to Host	
Bit 6-5	00 = Standard	01 = Class	10 = Vendor	11 = Reserved
Bit 4-0	0000 = Device All other codes are Reserved	00001 = Interface	00010 = Endpoint	00011 = Other

Figure 2-14. Format of a PC host request

The **RequestType** parameter is a bit field with bit 7 indicating the direction of the transfer for the next transaction. Bits 6 and 5 are a top-level switch of how the **Request** byte at offset 1 should be interpreted—this chapter covers only Standard Requests and Class Requests for a hub device. Other class requests are covered in later chapters. The lower four bits specify the destination of this request; the PC host can send requests to the device, to an interface, or to an endpoint.

The Request parameter indicates which information the PC host wants. Table 2-3 lists the standard requests for a device along with the required action by the I/O device. Table 2-4 is a similar list for a hub device.

Table 2-3. Standard PC host requests of an I/O device

Request	Required Device Action
Get_Status	Return current status
Clear_Feature	Clear Specified Feature
Set_Feature	Set Specified Feature
Set_Address	Store Unique USB Address and use from now on
Get_Descriptor	Return requested descriptor
Set_Descriptor	Set Specified Descriptor
Get_Configuration	Return Current Configuration or 0 if not configured
Set_Configuration	Set Configuration to the one specified
Get_Interface	Return Current Interface
Set_Interface	Set Interface to the one specified
Sync_Frame	Synchronize USB Frame Numbers (Async device)

Table 2-4. PC host requests of a hub device

Type	Request	Required Device Action
Standard	Get_Status	Return current status
Standard	Clear_Feature	Clear Specified Feature
Standard	Set_Feature	Set Specified Feature
Standard	Set_Address	Store Unique USB Address and use from now on
Standard	Get_Descriptor	Return requested descriptor
Standard	Set_Descriptor	Optional: Set Specified Descriptor
Standard	Get_Configuration	Return Current Configuration or 0 if not configured
Standard	Set_Configuration	Set Configuration to the one specified
Standard	Get_Interface	Optional: Return Current Interface
Standard	Set_Interface	Optional: Set Interface to the one specified
Standard	Sync_Frame	Optional: Synchronize USB Frame Numbers
HUB Class	Get_Bus_State	Optional (for diagnostics): Return D+, D–
HUB Class	Get_Hub_Status	Return Hub Status with Changed Identified
HUB Class	Get_Hub_Descriptor	Return Hub Descriptor
HUB Class	Set_Hub_Descriptor	Optional: Set Hub Descriptor
HUB Class	Set_Hub_Feature	Enable a standard hub feature
HUB Class	Clear_Hub_Feature	Disable a standard hub feature
HUB Class	Get_Port_Status	Return Port Status with Changed Identified
HUB Class	Set_Port_Feature	Enable a standard port feature
HUB Class	Clear_Port_Feature	Disable a standard port feature

The remaining 6 bytes of the PC host Request are used as data to support the request. If a single byte or word **Data Value** is required, it is supplied in byte offset 2 and/or 3. If a single **Index Value** is required, it is supplied in byte offset 4 and/or 5. The **Length** word in byte offset 6 and 7, if greater than 1, specifies the total length of the subsequent data transfer.

The optional second phase of a control transfer moves any required data to support the request between the PC host and the I/O device.

It is easier to understand these requests and their response through an example. In the next chapter we'll step through an enumeration sequence and identify all requests and responses.

Error Handling

A lot of care and effort went into creating a robust communications channel for USB. I have briefly mentioned some of the factors during this "Close to the wire" discussion. If we were building a USB transceiver, then I would go into much more detail at this point. The focus of this book, however, is **using** USB. There are many USB transceivers available (see a listing on the CD-ROM), and all of the implementations have been verified for correctness to the USB specification. If you are a silicon designer who needs to include a transceiver on your ASIC, refer to the reference design material in the Chapter 2/Silicon Design directory on the CD-ROM.

Differences for a Low-speed Device

A low-speed device can handle interrupt and control transactions; it does not support bulk or isochronous transactions.

The root hub knows the operating speed of each USB device, so the root hub knows when it is about to initiate a transaction with a low-speed device (how the speed is known is described in the next chapter). The root hub prefixes a special PRE token before all other tokens it generates for low-speed interrupt or control transactions to warn downstream hubs that a low-speed transaction is coming. Figure 2-15 shows a representative low-speed interrupt transaction. I won't describe this in detail because the I/O device itself does not see these PRE tokens and therefore does not need to deal with them. The hub acts on the PRE tokens by passing them through to all full-speed downstream ports and passing only the following token to low-speed downstream ports. Chapter 13 covers hub design in more detail.

One implication of adding the PRE prefix to interrupt and control transactions is that a low-speed device will never see a SOF token.

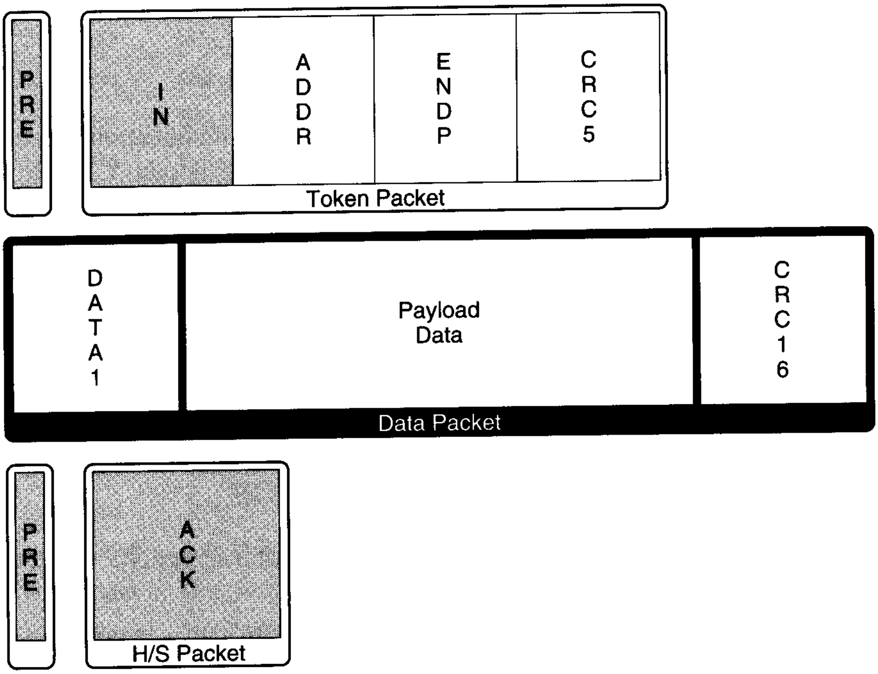
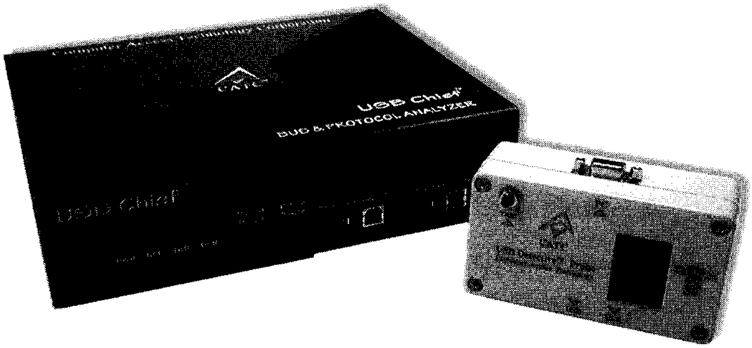


Figure 2-15. Low-speed packets are eight times longer

VIEWING THE USB PACKET BUS

It is interesting and educational to use an oscilloscope to look at the differential signals on the USB data wires, but this is not a productive tool to use in debugging a system. What we need is an observation tool that understands the packetized nature of the bus and can display the bus transactions at a higher level. Computer Access Technology Corporation has been working with USB since its inception, and their insight into the design challenges of a USB subsystem is obvious in their CATC tools (Figure 2-16).



Courtesy of Computer Access Technology Corp.

Figure 2-16. USB bus-specific tools from CATC

The CATC tools range from their Detective for USB observation to the Chief that can analyze, capture, replay, and run tests. All the tools capture activity on the USB bus and display it in the form of packet diagrams (Figure 2-17). Each token packet is displayed in a different color, and the sequence of packets that make up a transaction are grouped together.

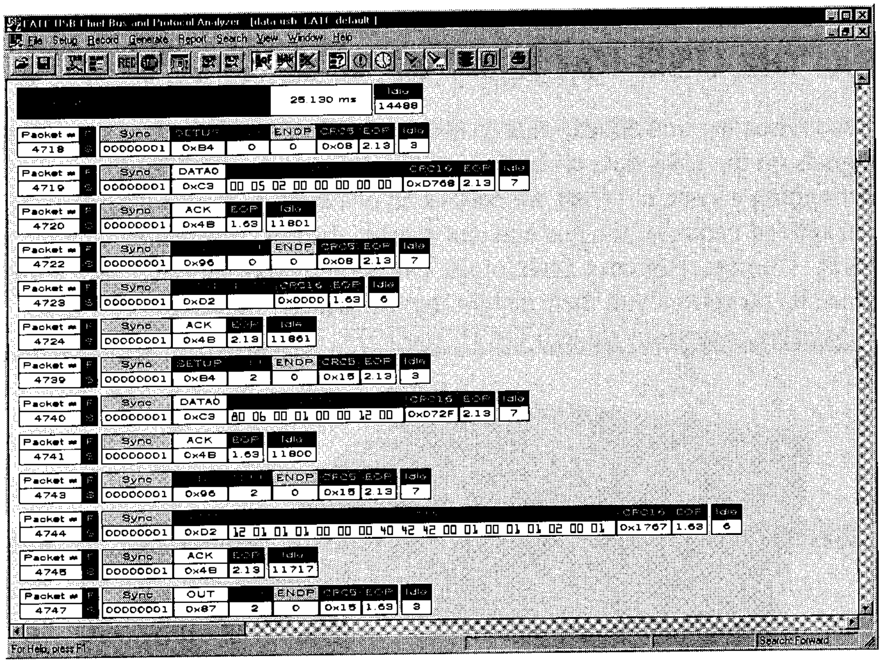


Figure 2-17. USB packets displayed as packets

All of the CATC tools capture bus activity for later analysis. The use of color to display different packet types and the grouping of building-block packets into transactions allow the designer to quickly interpret what has happened on the bus. The simpler CATC tools capture all bus activity while the more elaborate tools have programmable capture and programmable triggers. Being able to isolate packets sent to a specific device or triggering after, for example, device 42 has received 58 DATA0 packets, aids the debugging of more sophisticated USB devices. The high-end CATC tools can also be used to generate bus traffic for device reliability testing and failure analysis.

CHAPTER SUMMARY

This chapter provided insight into the signals on the bus, the fundamental packetized nature of the bus, and the transactions used to exchange data on the bus. The PC host uses a defined set of requests to control all of the devices attached to the bus, and these devices need to respond in a defined manner. Bus observation, or "sniffer" tools as they are called, are available to monitor and analyze these low-level bus signals.

CHAPTER 3
THE ENUMERATION PROCESS

Let us assume that the PC host meets all of the requirements described in Chapter 1, is running a USB-aware operating system, and has an available USB port. This port could be on the PC host itself (an embedded hub) or could be on an external hub. Now we have a new USB I/O device that we want to add to this running system. What actually happens between the host, the hub, and the device to deliver the many USB features?

After understanding what the PC host is doing, we learn the responsibilities of a general I/O device. All devices describe themselves using a set of descriptor tables. We start by looking inside the simplest example and then expand the discussion to cover the general case. We then derive the minimum hardware requirements for a device.

There are many "chicken-vs.-egg" situations in this section, so I'll need to defer some technical discussions to keep the flow of the concepts moving forward.

DEVICE DETECTION

Figure 3-1 shows details about the USB cable. The cable has four wires: two power wires for Vcc and Gnd and two signal wires for D+ and D–. The cable end that attaches to the hub has a Series A connector, and the cable end that attaches to the new device is either connected directly (no connector) or has a Series B connector. Both connectors have longer power and ground connector pins to ensure that the device has good voltages before signals are applied.

The hub socket supplies Vcc and Gnd. The current limiter will initially prevent more than 100 mA from being drawn, even instantaneously, from the hub. If excess current is drawn, then the hub informs the host software of this error (see "Enumeration steps," step 5), an error message is displayed on the PC screen, and the device is **not** configured.

Because we haven't plugged in the I/O device yet, it is in the **unattached** state.

In Figure 3-1, note the two biasing resistors in the hub; they ensure that D+ and D– are low when no device is plugged in. There is a single biasing resistor on the device that is attached to either D+ or D–. When the USB cable is plugged in, the biasing resistor causes D+ or D– to rise above ground, and this changed voltage difference is recognized by the hub. We have detected a cable being plugged in! By convention, if the device-biasing resistor is connected to D+, we are informing the hub that this device is full speed (12 Mbps), while a biasing resistor on D– indicates a low speed (1.5 Mbps) device. Simple and effective!

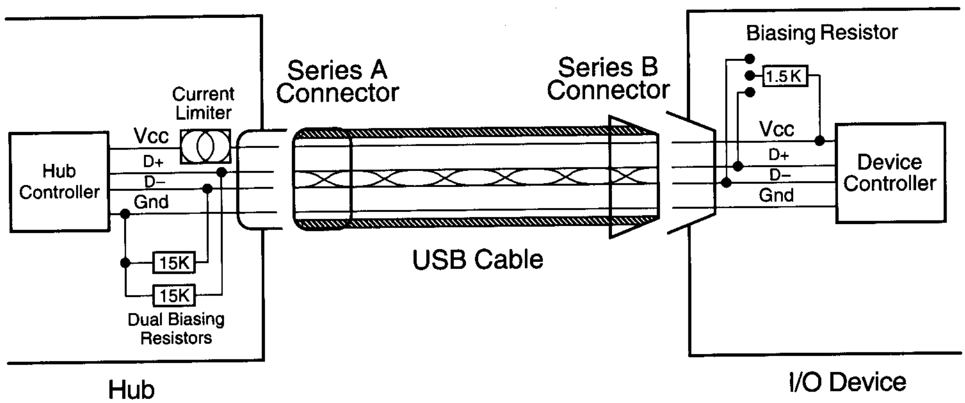


Figure 3-1. USB cable connection details

The I/O device is now in the **attached** state, and, because this hub is already configured and operational, the device moves to its **powered** state.

The hub updates a STATUS_CHANGE register following detection of the new device and then waits to be told what to do.

The PC host controls the enumeration phase and, during this phase, sends requests to two devices. The hub that identified the newly attached device will receive many requests for action, and the newly attached I/O device will also receive requests. If there are any other hubs between this hub and the root hub, they will not take part in this process. They will repeat the signals on the USB bus, because that is one of their roles, but because they are not being addressed by the PC host software during this process, they may be ignored.

The PC host software regularly polls all connected hubs looking for work to do. In most cases a hub has nothing to report so will NAK this interrupt transfer. But **this** time the hub responds with the STATUS_CHANGE data indicating which port has had a change in status—the PC host-based enumeration process has begun!

ENUMERATION STEPS

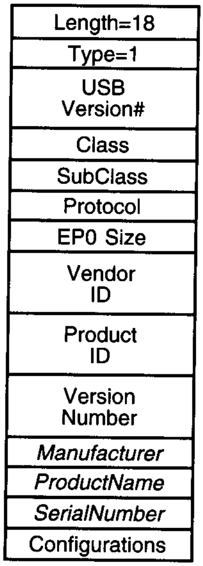
In the following description the PC host is initiating all requests. I have used a **ToHub:** prefix if the addressed device is the hub, or a **ToIO:** prefix if the addressed device is the newly attached I/O device.

1. **ToHub:Get_Port_Status**: Host discovers newly attached device.
2. **ToHub:Clear_Port_Feature(C_PORT_CONNECTION)**: Clears the flag in the STATUS_CHANGE register that started this process.
3. **ToHub:Set_Port_Feature(PORT_RESET)**: The hub responds by sending a reset to the I/O device. The hub maintains this reset for a minimum of 10 milliseconds. It then updates the RESET_CHANGE bit in its PORT_CHANGE register and enables the port for USB traffic by setting the PORT_ENABLE bit in the PORT_STATUS register. The PORT_CHANGE register update causes an update to the STATUS_CHANGE register. The PC host will notice this on its next scheduled poll.
4. **ToHub:Get_Port_Status**: The PC host discovers that the reset is complete.

5. **ToHub:Clear_Port_Feature(C_PORT_RESET)**: Clears the flag in the STATUS_CHANGE register. At this time the I/O device has power and has been reset; therefore, it is in its **default** state. The device will now respond to PC host requests to device address 0 (the default address). If **another** device were attached at this same instant in time, the PC host software will defer servicing it until the enumeration sequence on the **current** I/O device is completed. In other words, there will be only one device that responds to device address 0 at any one time.
6. **ToIO:Get_Device_Descriptor**: The PC host makes its first move to discover what kind of device has been attached. The device responds with a device descriptor (Figure 3-2). This data structure is described in the next section.
7. **ToIO:Set_Address**: The PC host allocates a device address to the newly attached I/O device. All subsequent requests are sent to this new device address. The device is now in its **addressed** state.
8. **ToIO:Get_Device_Descriptor**: The PC host repeats this request to the new address. It should get the same response shown in Figure 3-2. Any different response, or a device timeout, indicates an error condition that the PC host software must deal with.
9. **ToIO:Get_Configuration_Descriptor**: The device driver starts collecting information about the device, its interfaces, and its endpoints. In a feature-rich I/O device, the configuration can be quite extensive (for a preview, see Figure 3-15). For now, we will review the simpler example in Figure 3-3. All of the parameters will be explained later in this chapter. If the device has multiple configurations, then the driver typically reads them all. After some processing that may (hopefully not) require some user interaction, the host software will move to the next step.
10. **Select Device Driver**: From the PC host's perspective, it needs to determine which device driver is needed to support this newly attached I/O device. The algorithm for choosing a device driver is covered later. If the selected device driver is not currently in memory, it is loaded now.
11. **ToIO:Set_Configuration**: The device is now configured and operational, so it moves into its **configured** state.

DEVICE DESCRIPTOR

Figure 3-2 shows the device descriptor that is returned following the PC host's Get_Device_Descriptor request. We'll investigate each parameter but will use default values during this early discussion.



Note: Fields in *italics* are indexes into a STRING descriptor.

Figure 3-2. Device response to GET_DEVICE_DESCRIPTOR

The first three values, **Length** of this descriptor, **Type** of this descriptor, and **Version** of the USB Specification that we are compliant to, are easy to fill in (18, 1, and 100H, respectively). The **Class, Subclass,** and **Protocol** need some explanation.

Classes were created within the USB Specification to make it easier to write PC host software. In some cases, as we shall see, we do not need to supply **any** host software. Classes are somewhat confusing and appear unnecessary to the hardware designer, but because much of USB revolves around them, we will have to learn this subject! So what are the benefits of classes to the hardware designer?

The host operating system groups the diverse world of I/O devices into classes: I/O devices with similar characteristics are grouped together, and a generic device driver is provided for this class. Some examples include "Human-Interface-Device," "Audio," "Mass Storage," and "Power Supplies."

A subclass is a finer granularity grouping within a class. Protocol is used in some class definitions. So, if we can describe the I/O device within one of the predefined classes, the operating system will recognize the I/O device and will already have a software interface (that is, device driver) embedded in the operating system to converse with the I/O device. This means that we don't have to supply any Windows 98 code, any iMac code, any Linux code, or whatever to enable our I/O device to operate with these systems. A good point to note here is that USB devices can be used on all systems that support a USB port and these class drivers: An Intel PC keyboard operates on an iMac computer, and an iMac mouse operates on an Intel PC (aren't standards wonderful!).

If we are building a unique I/O device that the host operating system will not know about, we must supply a device driver, or *n* device drivers, one for each OS that we want to support. This is a lot of work, and we will avoid this in all the examples in this book. (My next book will include the gory details of device driver programming.)

There are two special class codes: 0 indicates that the class code is defined in a later descriptor; 0FFH indicates a vendor-supplied class code (this is even more work and beyond the scope of this book).

All other codes describe a USB Specification Class Type—this list is constantly growing, and an on-line reference is kept on www.usb.org. I have provided more discussion about device classes later in this chapter.

EP0 size defines the maximum number of bytes that can be sent or received by endpoint 0 in each data payload (the control endpoint that is always open). This is implementation-dependent, and many vendors have chosen the minimum USB Specification value of 8.

To sell USB products in the open market, you must have a **Vendor ID**. These are issued by admin@usb.org for a small registration fee. For our examples we'll use 04242H, which is my Vendor ID. We choose whatever values are appropriate for **Product ID** and **Version#**.

The next three values are **indexes** into an array of strings. We'll use real values in our later examples, but for now we'll use 0 for the default values.

The final value of **Configurations** defines the number configurations that this device has. Let us assume for now that this is 1.

To recap enumeration step 6 (and 8), the PC host discovered that the device was a conforming I/O device that it should talk to. Any illegal values in the device descriptor would cause the PC host software to ignore the device.

In enumeration Step 9, the PC host requests a configuration descriptor. Figure 3-3 shows the minimum example—this is for a device that has one configuration with a single interface that uses only endpoint 0 to exchange data. Endpoint 0 is predeclared, and, because it always exists, it need not be redeclared.

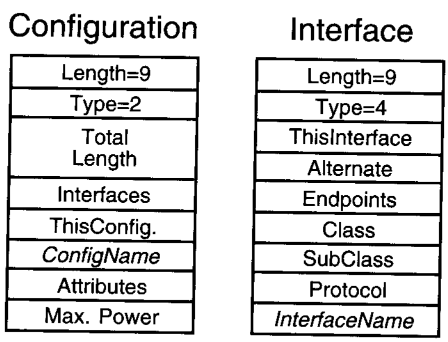


Figure 3-3. Descriptors for a minimal I/O device

Configuration Descriptor

A configuration descriptor has a fixed length and type: 9 and 2, respectively.

When the PC host requests the configuration descriptor, the device responds with all the descriptors that it uses to describe the functionality and operation of the device. These descriptors are concatenated together and sent as a single large block to the PC host. The **TotalLength** of this block is entered at byte offsets 2 and 3.

The number of **interfaces** supported by this configuration is entered at byte offset 4. A minimum I/O device will enter a 1 here.

A device with multiple configurations will have multiple configuration descriptors (we'll discuss this in the next section). Each configuration descriptor must have a unique identifier, called **ThisConfig**, and this is entered at byte offset 5.

The byte at offset 6 is an **index** into a string array. Because we aren't using these yet, we enter a 0 for this entry.

Configuration **Attributes** is a bit-mapped field with bits 7, 6, and 5 defined as shown below and bits 4 through 0 reserved (set to 0):

- Bit 7 set indicates that this device is powered by the USB cable.
- Bit 6 set indicates that this device has its own power source.
- Bit 5 set indicates that this device can generate a remote wakeup.

We will use 50 in the **MaxPower** entry to indicate that we need only the 100 mA that was initially provided. With this value, we would be considered a low-power device. We could request up to 500 mA from the bus, and some of the later examples will need this. Current requirements over 500 mA demand a self-powered device that includes a "wall-bug." A device can use both USB power and self-power. It is common to power the USB interface from the bus and use an external source for other device power. If the device power levels can be kept below 500 mA, then the device will not need an external power source. This will decrease the cost of the device and make it easier to use. It is worth the design effort to come in under the 500 mA limit. A device that needs to generate a REMOTE_WAKEUP will need external power to be able to generate this signal.

Interface Descriptor

A minimum device will have a single interface descriptor. The **length** and **type** are fixed at 9 and 4, respectively. We will enter 0 in the **Interface**, **Alternate**, and **Endpoints** fields, because they are not used in a minimum device.

A PC host device driver is bound to this interface descriptor. A device that supports multiple interfaces will have a device driver bound to each interface. It is important to realize that the PC host software views USB as a large collection of SOFTWARE INTERFACES—the I/O device is a convenient hardware implementation and container for these software interfaces. Although there is a matching software driver for each interface descriptor, multiple, similar interface descriptors can be supported by a single driver. The **Class**, **Subclass,** and **Protocol** entries determine which software driver will be bound to this interface.

We use 0 for the **InterfaceName** because we are not yet using strings.

CHOOSING A DEVICE DRIVER

The Windows 98 driver selection scheme gives you an extreme amount of control. You can use the Vendor ID, Product ID, and Device Revision number to pinpoint a driver that exactly matches the hardware. You will, of course, need to **write** this device driver yourself because it will be unique to the I/O device.

The strategy I follow in this book is to make use of the device drivers already installed in the operating system. This will mean that we will **not** need to write our own device driver—this is a specialized task that requires a certified wizard to complete. Let's look at the nonwizard methods we can use to choose a device driver.

The key to our solution is to request a device driver that Windows already has. We expect the search for a driver based on Vendor ID, Product ID, and Device Revision to fail, because we have not supplied a device driver that matches those elements. Once the search fails, the PC host software looks at the Interface Class, Subclass, and Protocol. The operating system supports an array of class drivers, and for our first set of examples we declare our I/O device as belonging to the Human Interface Device, or HID, class.

Mice, keyboards, and gamepads are good examples of HID devices. They react very slowly (when compared with a 500-MHz Pentium III processor) and transfer only small amounts of information between the PC host and the I/O device. The information transfer for USB HID devices is further standardized using **Reports**. A report is a preformatted data structure that defines how much data is exchanged and how this data should be interpreted. This logical representation allows the I/O device to be radically changed providing it maintains its Report interface. This structure gives hardware independence to the software, enabling both to be improved on different schedules. The details with a Report are described later in this chapter. Our first example in Chapter 6 uses a report for data exchange.

A device with the minimum descriptors just described would enumerate correctly, and a custom device driver would have been loaded to support the device. Let's look at some additional entries we need to make in the descriptor tables so that we don't have to supply our own device driver. How does a mouse or keyboard enumerate so that the operating system uses one of its internal drivers? These devices are Human Interface Devices (HID), and the operating system knows a lot about these! The operating system supports many other classes; I'll present these in later chapters when appropriate examples require them.

DEFINING AN HID

An HID-class device is identified by a class code of 9 in the interface descriptor. This entry will prompt the PC host to look for an HID Descriptor and a Report descriptor as shown in Figure 3-4.

These would have been concatenated with other descriptors and supplied to the PC host following a Get_Configuration request.

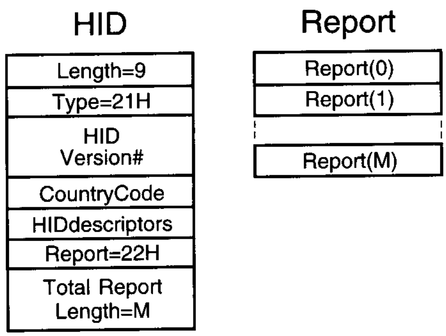


Figure 3-4. HID and report descriptor formats

HID Descriptor

Many of the examples in this book are human interface devices, so we need to slow down here to absorb all of the information in this section. Although HID operation sounds difficult at the first reading, it will turn out to be the easiest option to use to connect a unique I/O device to the PC host.

The **HID** class consists primarily of devices that people use to control the operation of computer systems. Typical examples include:

- Keyboards and pointing devices: mouse devices, trackballs, joysticks
- Front panel controls: knobs, switches, buttons, sliders
- Controls that might be found on devices such as telephones, VCR remote controls, games or simulation devices: data gloves, throttles, steering wheels, pedals
- Devices that may not require human interaction but provide data in a similar format to HID-class devices: bar code readers, thermometers, voltmeters

Many typical HID class devices include indicators, specialized displays, audio feedback, and force or tactile feedback. Therefore, the HID class definition includes support for various types of input and output directed toward the user.

The first four entries of an HID descriptor have standard values: 9, 21H, 100H, and 0. If the HID device operates only in a single country, then **CountryCode** should be set appropriately (for more information, see Section 6.2.1 of USB Documents/HID Class Definition.pdf on the CD-ROM). A keyboard would be a good example of a country-specific I/O device.

The fifth entry, **HIDDescriptors**, contains a count of the number of HID class descriptors that follow. A single I/O device could have several HID interfaces.

The sixth entry is fixed at x22H indicating that this interface contains a **report descriptor,** and the seventh entry contains the **TotalLength** of this report descriptor.

Report Descriptor

The primary and underlying goals of the Report Descriptor are:

- Be as compact as possible to save device data space.
- Allow the software application to skip unknown information.
- Be extensible and robust.
- Support nesting and collections.
- Be self-describing to allow generic software applications.

A type of pseudocode has been generated that is able to generically describe any style and quantity of data (in Chapter 5, I'll describe an HID tool that does the work for you). The coding was chosen to be very compact and to make it easy for the PC host to parse and generate. As a result, it is a little difficult to read, as evidenced by the keyboard descriptors in Figure 3-5. I have added comments to help you decipher the report. This report generates a 3-byte buffer that is sent to the keyboard to illuminate the status LEDs; from the keyboard, the report receives a 6-byte buffer that can contain up to six keystrokes. Reports are introduced in the next chapter, but for a full discussion of the Report structure, see the USB documentation directory on the CD-ROM.

A Report Generator tool is provided on the CD-ROM to help build custom reports. All reports used in our initial HID examples use the same buffer format: A byte-wide buffer of up to 64 entries can be sent to the I/O device, and the I/O device can supply a similar buffer. Each example will individually interpret the byte fields.

```
Usage Page (Generic Desktop),          ;Use the Generic Desktop
Usage (Keyboard),                      ;Start Keyboard collection
        Collection (Application),
        Usage Page (Key Codes),
        Usage Minimum (234),
        Usage Maximum (231),
        Logical Minimum (0),
        Logical Maximum (1),           ;Fields values from 0 to 1
        Report Count (8),
        Report Size (1),
                                       ;Add fields to the input report.
        Input (Data, Variable, Absolute),
        Report Count (5),              ;LED Report = 5 LEDs
        Report Size (1),               ;Size of each report item is 1 bit
        Usage Page (LEDs),
        Usage Minimum (1),
        Usage Maximum (5),
        Output (Data, Variable, Absolute),
        Report Count (3),
        Report Size (1),
        Output (Constant),             ;3 bits of Padding
        Report Count (6),              ;6 characters are buffered
        Report Size (8),               ;Characters are byte wide
        Logical Minimum (0),
        Logical Maximum (101),
        Usage Page (Key Codes),
        Usage Minimum (0),
        Usage Maximum (101),
        Input (Data, Array),           ;6 character buffer
        End Collection,
End Collection
```

Figure 3-5. Commented keyboard report

I/O Device's Point of View

We have already seen that the I/O device exists in various states—disconnected, attached, powered, etc. Figure 3-6 shows the complete state diagram that an I/O device must adhere to. A new state, called **suspended**, is important for correct I/O device operation and is introduced here.

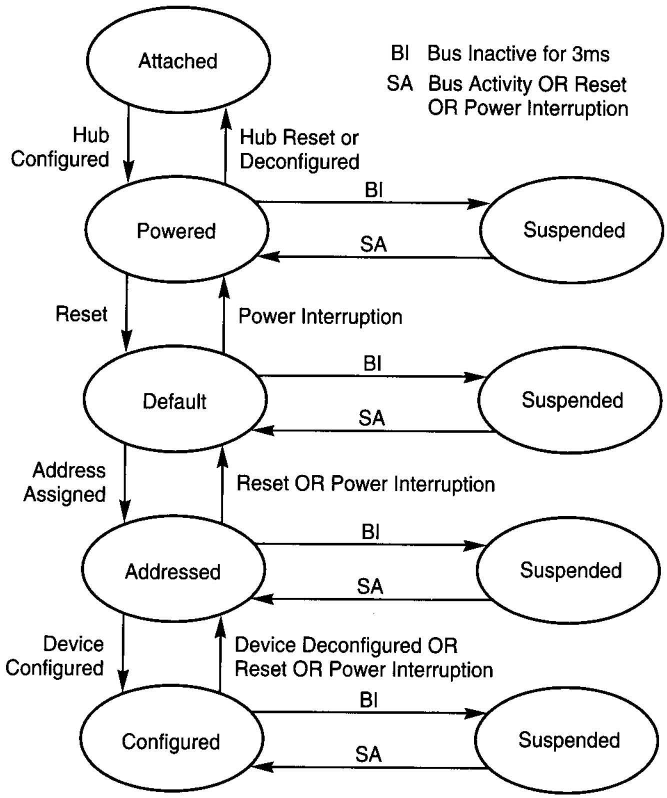


Figure 3-6. Required I/O device state diagram

If an I/O device detects no bus activity for 3 ms, the device is **required** to move to a low-power **suspended** state, in which it draws no more than 0.5 mA from the bus. No bus activity for 3 ms means that the PC host has stopped sending SOF packets; this is a result of the PC host powering down. There's no point in keeping most I/O devices powered on if the PC host has powered down! Any activity on the bus will result in the I/O device returning from a suspend state into one of the active states.

It is possible for the I/O device itself to bring the PC out of its powered-down state. This capability, called **REMOTE_WAKEUP**, is the **only** time a signal is initiated by the I/O device. If the I/O device were a telephone, for example, it would want to wake up the PC if the phone rang. This capability must be predeclared in the device configuration descriptor so that the PC host knows to prepare for such an event.

The I/O device drives a remote wakeup signal (a Kstate onto the idle bus) to alert its local hub. Hubs propagate this signal up to the root hub, which informs the PC to wake up.

It should be noted that the PC host is allowed to send control requests even after the I/O device is in the configured state. Indeed, the PC host could send a Set_Configuration request specifying the zero configuration that moves the I/O device back to the addressed state where it stops responding to the real world.

Enumeration from the device point of view is straightforward (Figure 3-7). Requests are received from the PC hosts, and the I/O device must respond to them. The I/O device designer predeclares all of the descriptors and supplies the correct information to the PC host on request. We will work through several design examples in Chapter 6.

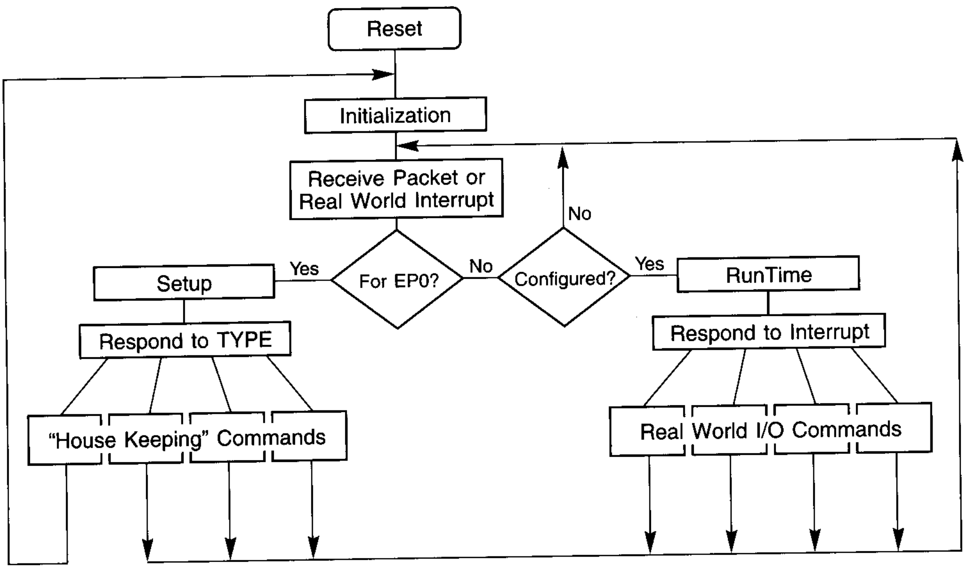


Figure 3-7. Typical software structure of an I/O device

MINIMUM I/O DEVICE

With multiple states to operate in and with control and data packets to respond to, we can agree that even a minimal USB I/O device is more than "a few PALs." Figure 3-8 shows a block diagram of the essential elements of a USB I/O device.

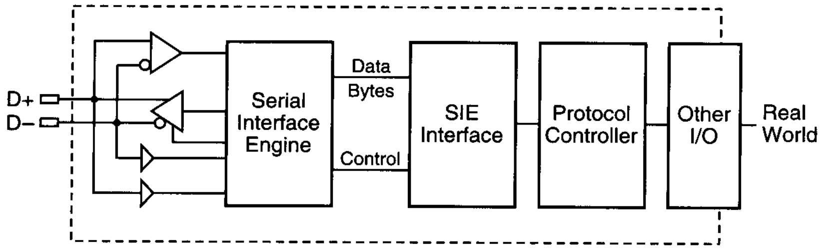


Figure 3-8. Essential elements of a USB I/O device

A USB transceiver is required to meet the electrical characteristics of the bus. These electrical requirements are generally integrated into a USB device controller, but, if required, Philips has a PDIUSBP11 component (Figure 3-9).

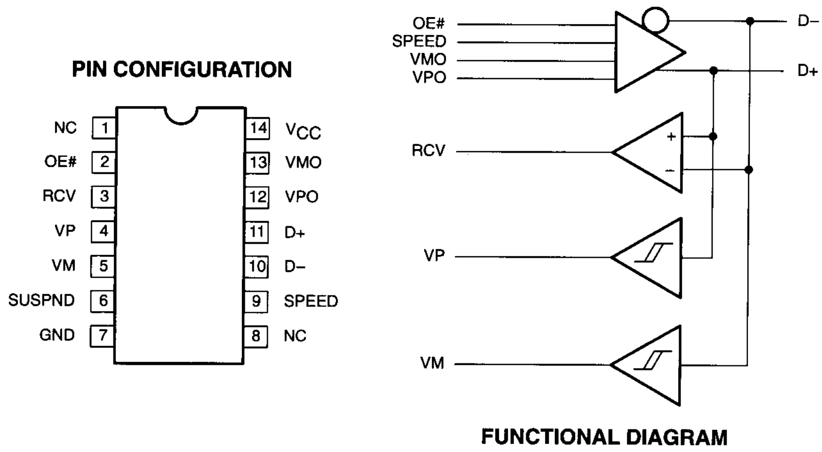


Figure 3-9. Stand-alone USB transceiver

An essential part of a USB interface is the Serial Interface Engine, or SIE. The SIE receives bits from the USB transceiver, validates them, and provides valid bytes to the SIE interface. Similarly, bytes are received from the SIE interface and transmitted serially onto the USB bus.

The SIE contains intelligence to manage the bus bit-level protocol including the bit-stuffing algorithm. A vendor can include more intelligence in the SIE if desired or can pass raw data with possible error status to the protocol controller. The richness of the SIE interface also varies with different vendors—some supply the bytes from USB in FIFOs, some in memory; some supply error status flags and expect the protocol controller to deal with error conditions, while some error-check the incoming data, implement the USB handshake protocol, and only interrupt the protocol controller once clean, validated data has been received. We'll use a variety of USB components in the examples and will get first-hand experience of these implementation differences.

Several stand-alone SIE components are available (see the CD-ROM for a full list), and these vary in their SIE interface. Two popular interfaces are the parallel port and I^2C serial interface as shown in Figure 3-10.

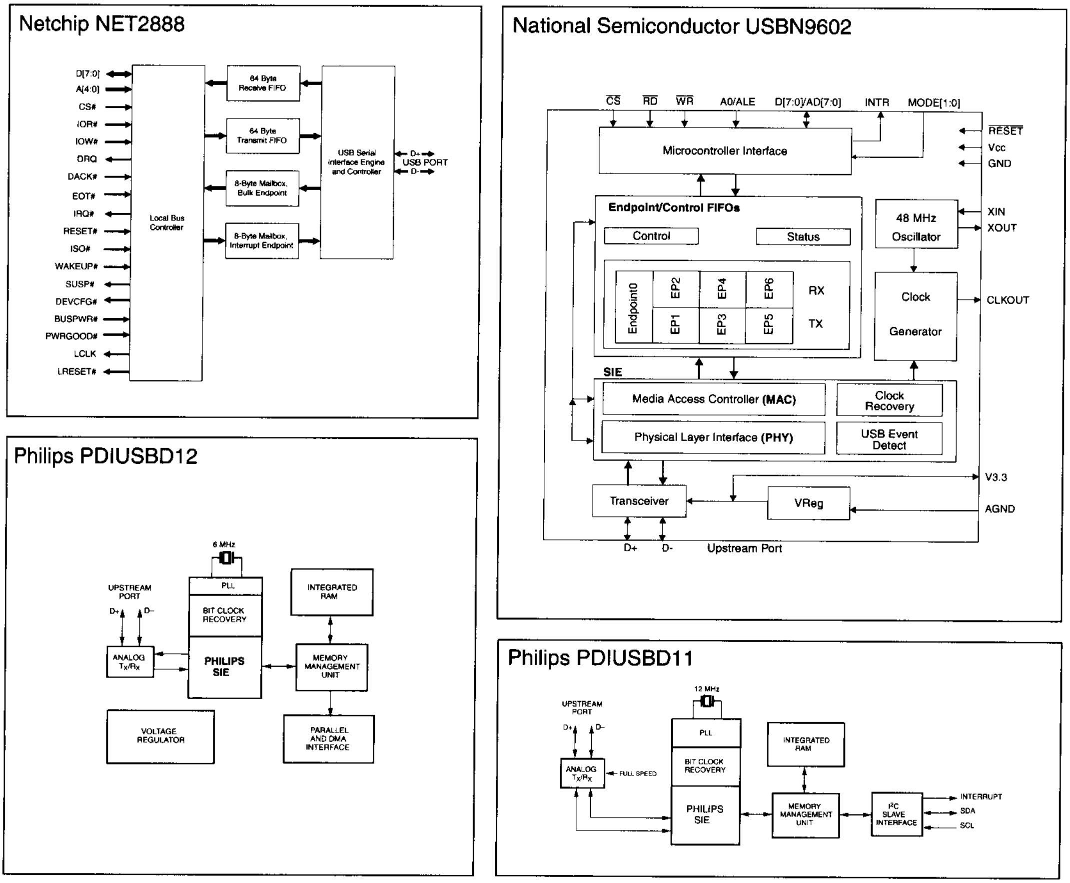


Figure 3-10. Stand-alone SIE peripherals

All SIE components have the same structure because they all interface to the same USB bus definition. Implementations differ depending on the number and type of endpoints that are supported and the supported speed of connection, full at 12 Mbps or low at 1.5 Mbps.

The Protocol Controller can be implemented in a variety of ways, and the most common is a programmable microcontroller. Many microcontroller suppliers have implemented a SIE into their family of devices to create a single-chip USB controller. Different vendors use different microcontrollers so different instruction sets are used. There is a large span of program memory space varying from 0.5K to 32K. Most vendors produce a family of devices so you can better match your requirements. The microcontroller's program memory may be ROM, EPROM, or even EEPROM. If ROM is used, then an external, serial EPROM is typically used to customize a reasonably fixed I/O device configuration. EPROM devices are programmable, and some versions are shipped in a plastic package for lower cost—these sealed devices are not erasable and are referred to as One-Time-Programmable (OTP). An EEPROM, sometimes called a flash device, is programmable in-circuit. These differences will become clear once we discuss the development environments in Chapter 5.

For microcontrollers, there is an even wider diversity of implementations available, ranging from a simple 8-bit implementation such as the Cypress 63100 device to the 32-bit Motorola 823 device (Figure 3-11).

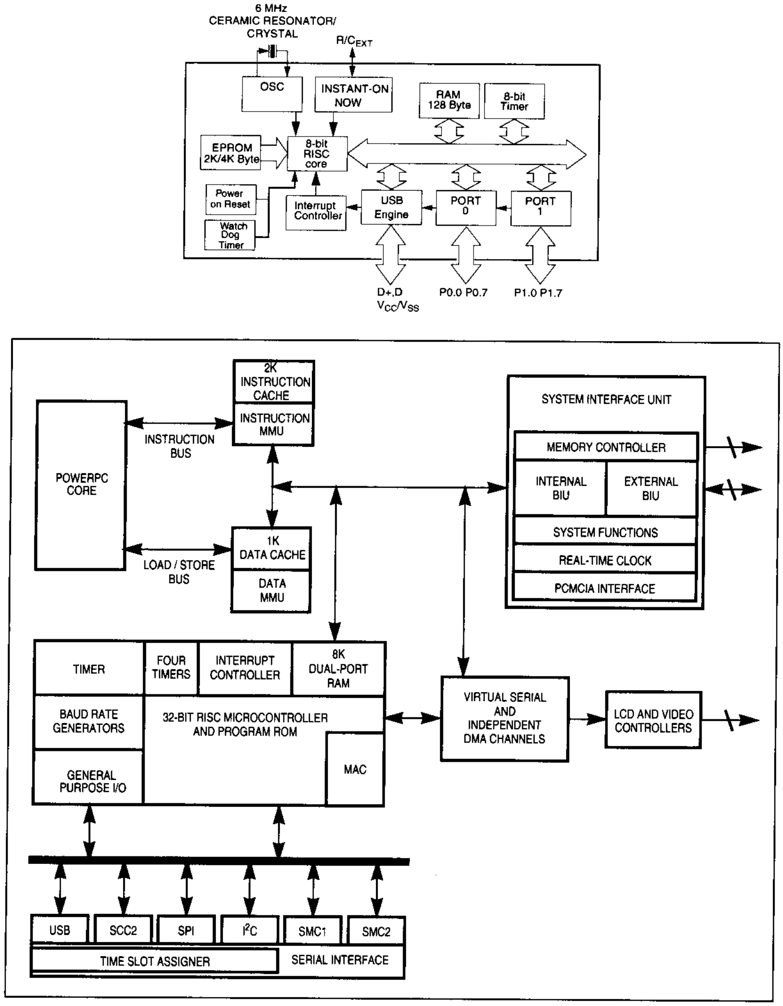


Figure 3-11. Wide range of USB microcontrollers available

The density and complexity of the integrated I/O capability also vary greatly. It is best to design your I/O device first, and then look for the best match of features from a list of available components (see Appendix A for a selection guide).

One device we'll use in several examples is the EZ-USB component from Anchor Chips (Figure 3-12). This component has a unique implementation in that the program memory of the protocol controller is RAM. On power-up, an intelligent SIE holds the protocol controller in reset. The SIE understands the enumeration process and can complete the process without help from the protocol controller. The device driver specified by the EZ-USB device descriptor knows how to do one thing: Download a program into the program RAM and remove the reset from the protocol controller. We thus soft-load a program into the I/O device!

The EZ-USB component then programmatically detaches itself from the hub and reattaches itself with its newly loaded personality. Anchor Chips calls this process "renumeration," and it means that the device doesn't have to be built with a mask ROM, be programmed, or even be flashed. If you think that your I/O device program may change after it ships to users, then software update is as easy as providing a new file on the PC host. No product to recall. No new parts or EPROMs to manufacture and supply. Just ask your users to download an update from the Internet!

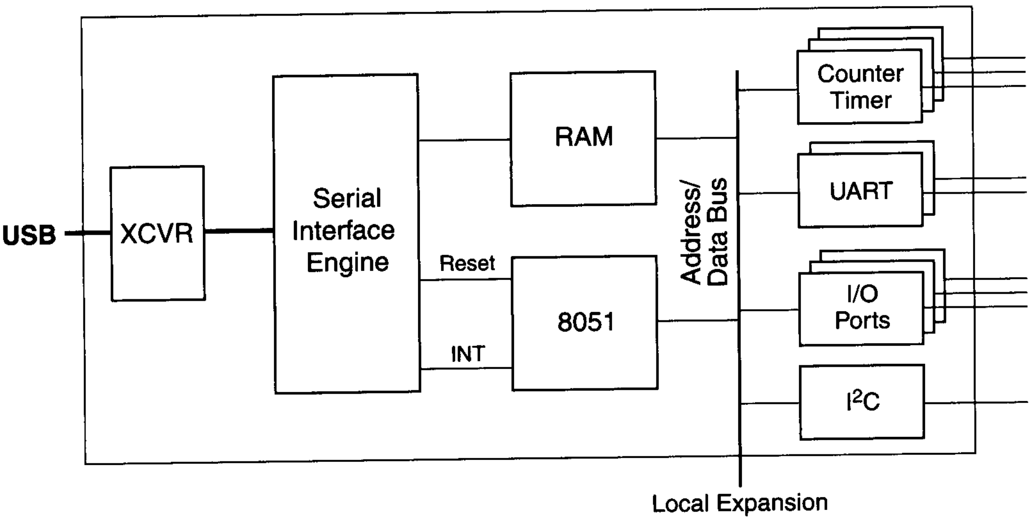


Figure 3-12. EZ-USB component from Anchor Chips

The EZ-USB component is a little more expensive than ROM or EPROM parts because its program in RAM means that the die is larger. For low- and medium-volume projects, however, use of this component could be a cheaper solution overall. Anchor Chips has compatible ROM parts for high volume—these are not downloadable, but you are not penalized for designing a successful product (the part has a lower unit cost but fewer features). Because the many examples in this book have different personalities, I will use the Anchor Chips EZ-USB component for these examples.

Complex I/O Device

This section expands on the minimal descriptors described earlier in the chapter. Edit the descriptors slowly so you can appreciate their building-block nature.

First we'll add a **Strings** descriptor. This has a variable length header followed by variable length string entries (Figure 3-13). The header consists of a **length** and **type** entry (2N+2 and 3) followed by an array (N) of **LanguageIdentifiers.** Each string entry consists of a **length** and **type** entry followed by a unicode **string**. A unicode string uses a word to represent each character and is not NULL-terminated. For more information on unicode, please refer to *The Unicode Standard, Worldwide Character Encoding,* produced by The Unicode Consortium and published by Addison-Wesley, Reading, Massachusetts, U.S.

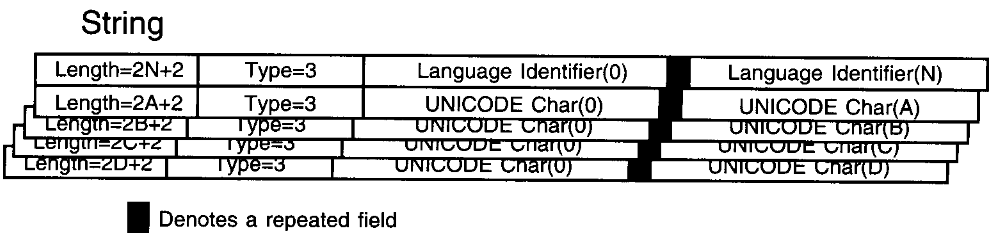


Figure 3-13. Format of a strings descriptor

For the examples here, I use a NULL in the high byte and an ASCII character in the low byte. The order in which the strings are declared will define their **INDEX**—the indexes start from 1 because a 0 is used to define "no string." We can now use useful, human-readable strings in our I/O device and go back to our descriptors and back-fill the string index entries. These strings are helpful during the debug and enumeration phases because they allow Windows to better identify the device (or it uses "unknown device," which is not very helpful).

Our minimal descriptors used the pre-existing endpoint 0 for run-time data exchanges. This is adequate for interrupt transfers, but endpoint 0 does not support bulk or isochronous transfers. If we need one or more of these types of endpoints, we declare them using endpoint descriptors (Figure 3-14).

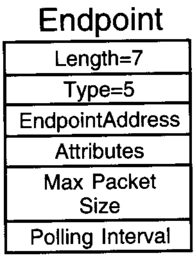


Figure 3-14. Format of an endpoint descriptor

The **length** and **type** of an endpoint descriptor are fixed at 7 and 5, respectively. If multiple endpoints are defined, each will need an endpoint descriptor that is uniquely identified by the **EndpointAddress** entry.

The endpoint address uses bits 3:0 to specify the endpoint number and bit 7 to specify if this is an IN endpoint (=1) or an OUT endpoint (=0). Bits 6:4 are reserved and set to 0.

The **Attributes** entry uses bits 1:0 to specify the endpoint type, control (=00), isochronous (=01), bulk (=10), or interrupt (=11). Bits 7:2 are reserved and set to 0.

The word at byte offset 4 specifies the **MaxPacketSize** that this endpoint can support.

The **Interval** entry is used by isochronous endpoints (must be set to 1) and by interrupt endpoints (set a value from 1 to 255) to specify the interval time, in milliseconds, of PC host polling.

A typical device has multiple interfaces and therefore multiple interface descriptors. Each interface descriptor specifies one or more endpoint descriptors required to implement this interface. Because these multiple interfaces are supported concurrently, each must specify exclusive use over its endpoints. That is, an endpoint cannot be referenced by more than one **active** interface descriptor.

An interface descriptor is **active** if it is specified in the **current configuration**. Some devices have more than one configuration, and only one configuration may be selected at one time. Each configuration may reference some of the interface

descriptors of other configurations, but these become active only if contained in the currently selected configuration.

The flexible scheme results in a collection of descriptors (Figure 3-15). The total length of a configuration descriptor can be quite long (remember, this is the concatenation of all the descriptors except the device descriptor and the strings descriptor).

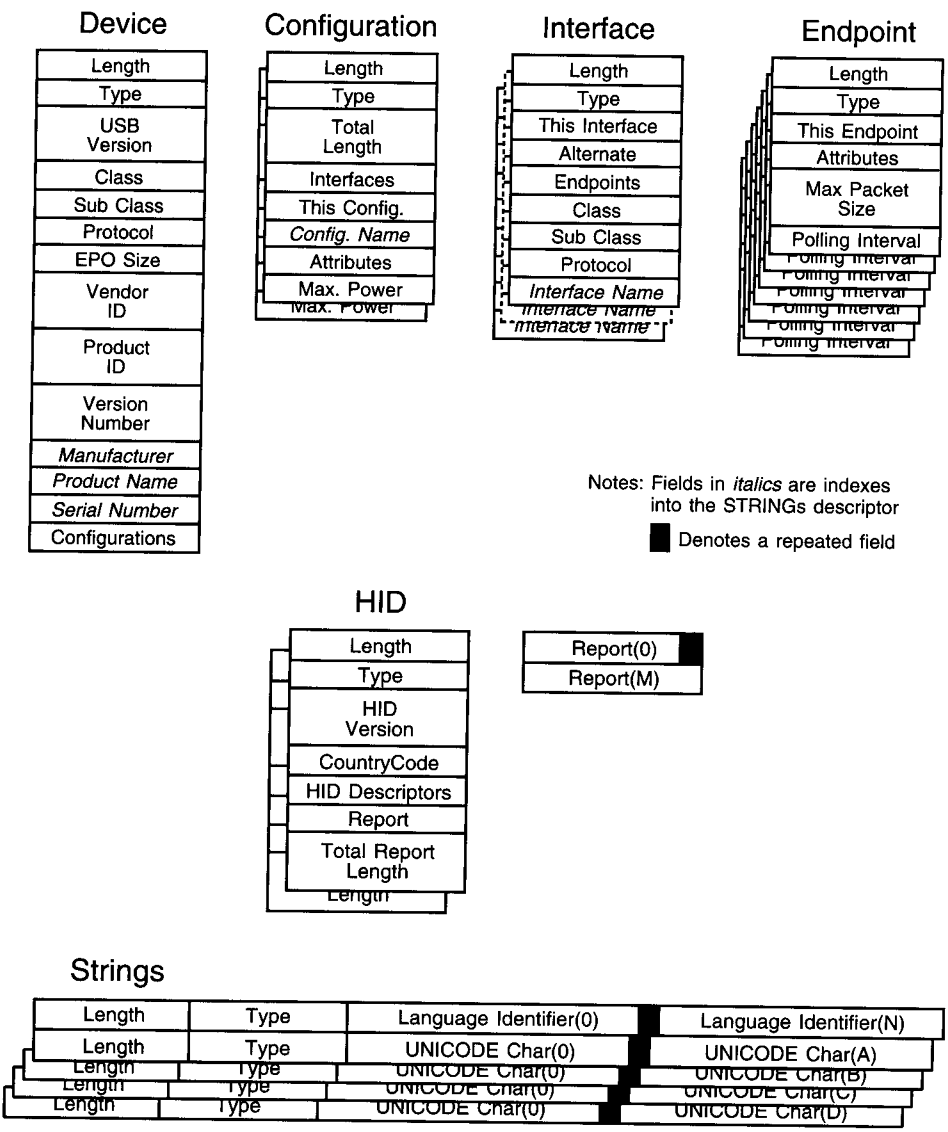


Figure 3-15. Descriptors for a complex I/O device

Chapter Summary

It has now been about 100 ms since the beginning of this chapter, and the PC host is ready to use the I/O device that we have added.

We saw how the enumeration phase was completed and learned the responsibilities of an I/O device. We stepped through the initialization and configuration information that was delivered to the PC host. The descriptors for a simple device are small, but the scheme allows for complex devices that can have several configurations, each supporting multiple interfaces. An I/O device requires a USB transceiver, a Serial Interface Engine, and a protocol controller. This is more than "a few PALs," but low-end USB controllers may even be cheaper than these PALs. A wide range of hardware exists to engineer a USB I/O device—a device can be selected to match the design task.

CHAPTER 4
RUN-TIME SOFTWARE ON THE PC HOST

In this chapter we'll write two example PC host programs, both of which will set us up for writing our application program in Chapter 6.

- **Example 1:** This will be a **USB Device Display** program that extracts and displays all of the descriptors, introduced in the previous chapters, from all of the devices currently attached via the USB bus. The program does this by searching for devices on the USB bus, so we'll learn valuable techniques for the "low-level" USB activity.
- **Example 2:** This will be an **HID Display** program that searches for and displays information on the currently installed human interface devices (HID). The program uses a high-level file-style access to the I/O devices so that we focus on the **information** transferred between the PC host and the I/O device and not on the details within the USB packets.

The final section of this chapter describes the standardized REPORT mechanism used to exchange data between the PC host and an HID. I will present the general approach first and then give a specific example that will be used in later chapters.

Before jumping into coding, we should discuss operating system support for USB. Figure 4-1 shows the layered support for USB in the Windows 98 operating system.

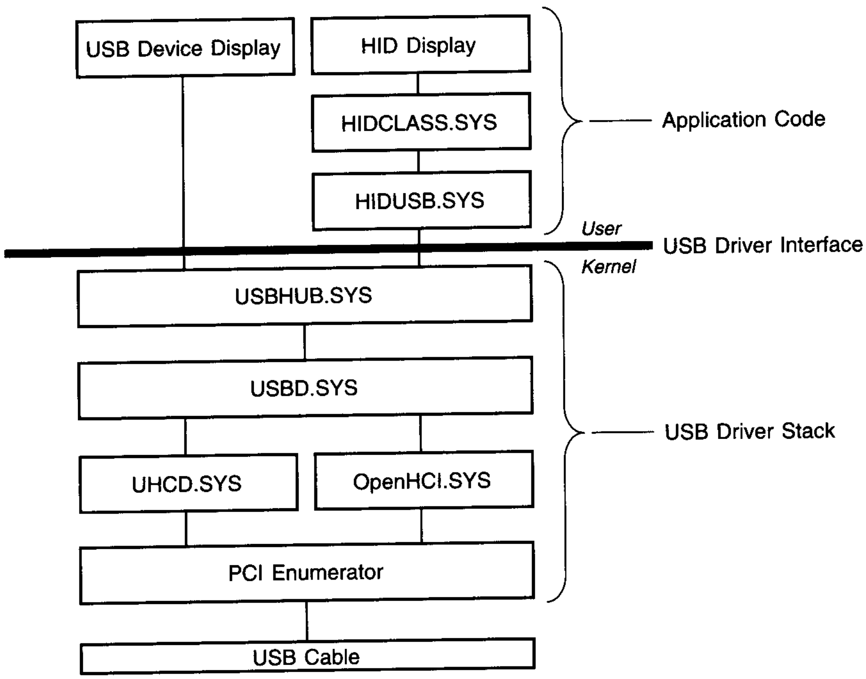


Figure 4-1. USB support in Windows 98 is layered

The USB Driver Interface of Windows 98 is part of the Win32 Driver Model (WDM) layered architecture. Our device display program will interface directly to the kernel-level USB driver stack, and our HID display program will interface with the HID Class driver stack. I have written the example programs in Visual Basic, which hides some of the complexities of writing a Windows application program. Use of Visual Basic also makes applications more portable if your target OS is not Windows. Both example programs make extensive use of system calls; these are declared in a separate module to allow the programs to be more easily understood. The source of both programs can be found in the Chapter 4 directory on the CD-ROM.

VISUAL BASIC REVIEW

I chose Visual Basic to allow the example programs to be easy to read and easy to understand. There are a few things about Visual Basic that I should repeat here for your convenience.

A Visual Basic program is built around a graphical human interface that uses objects such as buttons, graphics, and text (Figure 4-2). The visible window that you interact with during program construction is called a FORM, and a single program could include many forms to present information to the user. The user interacts with the program by clicking the mouse when it's over an object on the form or by entering text from the keyboard.

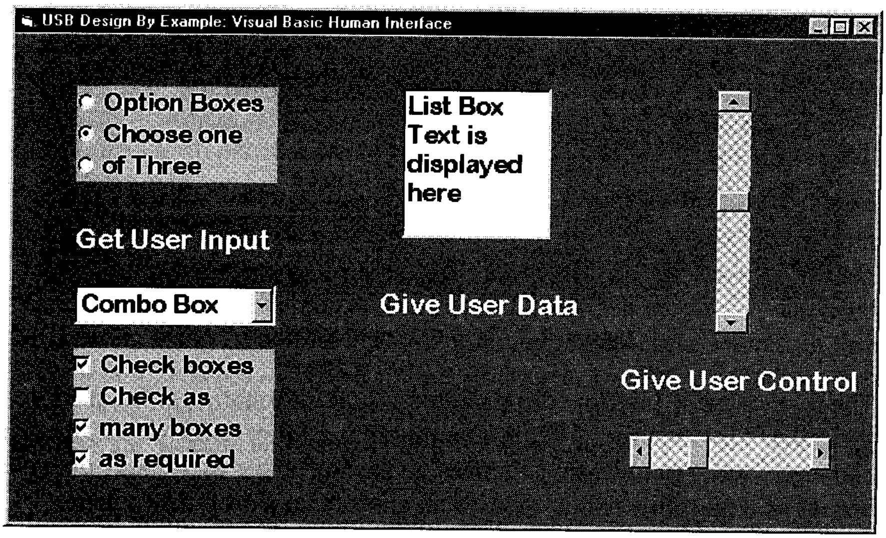


Figure 4-2. Visual Basic has a graphical human interface

Each object has many properties that can be set or tested. When one of these properties changes, Visual Basic first checks to see if you have defined an "action" subprogram, or script, to run in response to this interaction. A BUTTON object, for example, will be informed when the mouse is over it and if one of the mouse buttons is clicked. Visual Basic does all of the hard work for you, letting you focus on a few responses to actions on key visual objects.

Both example programs make extensive use of the ListBox feature of Visual Basic. A ListBox can be displayed as a multiline list (with scroll bars if required) or as a single line with the list contents available in a drop-down menu. Figure 4-3 shows both styles.

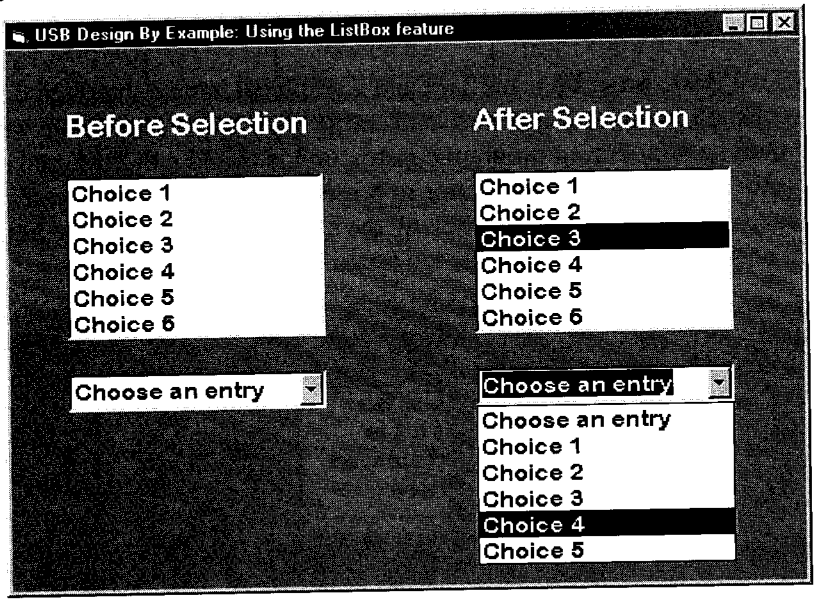


Figure 4-3. Two styles of ListBox

The two styles of ListBox allow related visual information to be quickly assimilated by the user. The programmer has a lot of control over the displayed elements, and Visual Basic generates an "event" if the user clicks on any of the entries of a multiline list or chooses an entry from a drop-down menu.

A ListBox has two data values associated with each line: The displayed text is called a **LIST (n)** value, and there is a hidden element called **ITEMDATA (n)**. This feature allows programmer-useful information to be coupled with user-useful information that, in turn, simplifies the example software.

Other key Visual Basic attributes are described briefly below.

Variable Names—The size of all variables is declared by adding a symbol after a variable's name:

- "&" means "32-bit integer"
- "%" means "16-bit integer"
- "$" means "a string"

There are others, but I have not used them in these example programs.

User-Defined Data Types—Complex data types can be constructed using the base data types supported by Visual Basic. We saw in the previous chapters that large data structures are used to describe the functions and features within USB. These data structures can be defined as user-defined data types, and this allows our software to be written more simply and clearly.

Parameter Passing—All of the OS libraries that we will be calling are written in C and expect parameters to be passed by VALUE. The convention that Visual Basic uses is to pass parameters by REFERENCE, i.e., the **address** of the variable. C programs often include pointers to variables, and the programmer can manipulate pointers as easily as manipulating data variables. Visual Basic does not allow this—using pointer arithmetic, it is as easy to write buggy code as it is to write efficient, elegant code! Careful declaration of the called libraries will implement any parameter conversions necessary. Where this parameter conversion could not be finessed via the function or subroutine declaration, I have used an "AddressFor" function supplied with Dan Appleman's *Visual Basic 5.0 Programmers Guide to the Win32 API* (ISBN 1-56276-446-2). This library is supplied, with permission, in the Chapter 4 directory on the CD-ROM.

System Call Parameter Values—Microsoft provides a lot of scope and flexibility with their operating system libraries. With this flexibility comes responsibility: If you pass an invalid parameter to one of these libraries, the results are not always predictable. I have limited the amount of damage that can be done by using "magic" numbers, typically hexadecimal values, in these calls. **DO NOT CHANGE THESE** unless, of course, you know what you are doing. Many books have been written on API programming, and my goal is not to explain the benefits and dangers of these functions but to USE THEM to implement USB-focused applications.

There are many textbooks available for Visual Basic, and I recommend getting a good understanding before proceeding further in this chapter.

EXAMPLE 1: USB DEVICE DISPLAY

Because USB supports the dynamic attach and detach of I/O devices, the address at which a particular device is located will vary according to the order in which it is attached and which other devices are already attached. Thus, we can't rely on a static address to locate an I/O device; instead, we must work with the operating system and gain access to its system tables. Windows 98 provides system calls to allow us to do this.

We must first locate the root of the Plug and Play I/O subsystem and then search for USB host controllers. USB host controllers have a predeclared name of "HCDx," where "x" is a digit and identifies a unique host controller. The OS can support multiple USB host controllers, and each can support up to 126 devices on its 12-Mbps shared bus. Systems that use several high-bandwidth I/O devices may have multiple host controllers, and these must be interrogated too.

Once a USB host controller is found, its corresponding root hub is identified. The ports of this hub are then queried to discover if a device or another hub is connected. If another hub is discovered, then the ports on that are queried. All hubs are queried recursively to the maximum depth of five as defined in the USB Specification. Figure 4-4 shows the algorithm used to create a list of attached USB devices.

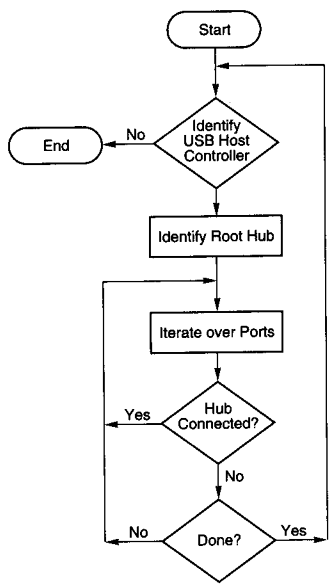


Figure 4-4. Search algorithm for USB devices

Example 1—Step 1: Human Interface Design

The first part of any Visual Basic program is the design of the user's view, or Human Interface, of the program. I chose to use two forms—the first to display the topology of the attached USB devices and the second to display the descriptors of a chosen device. Figure 4-5 shows the opening form for this example. It consists of four buttons labeled Host Controller 0 through Host Controller 3, a Status Line, and a Display area.

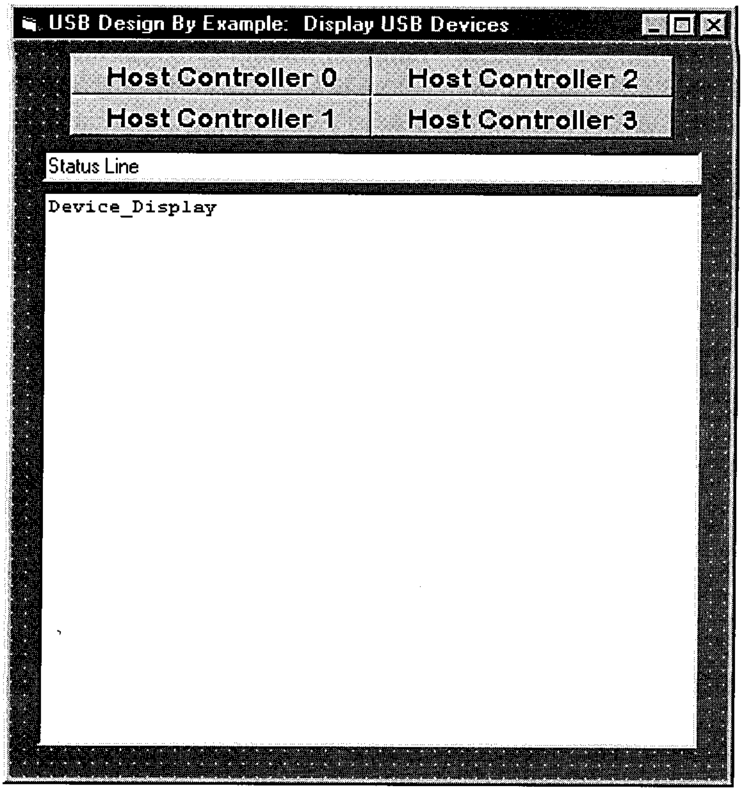


Figure 4-5. The first, and opening, form

Figure 4-6 shows the second form; it is designed to look as similar to the descriptor diagrams in the previous chapters as possible. All devices will have a **Device** descriptor and at least one **Configuration** and one **Interface** descriptor. A device may have one or more **Endpoint** descriptors and may have one or more **Class** descriptors (such as HUB, Human Interface Device, Communications Device). This human interface design allows one of each descriptor type to be displayed at a time; a CHOICE box above each descriptor type selects one if multiple descriptors are detected on this device. The CHOICE box also allows the names of each descriptor parameter to be displayed. As an aid to learning, if any parameter name of any descriptor is clicked, the program displays the corresponding parameter value. The final CHOICE box displays the **String** descriptors if present on this device.

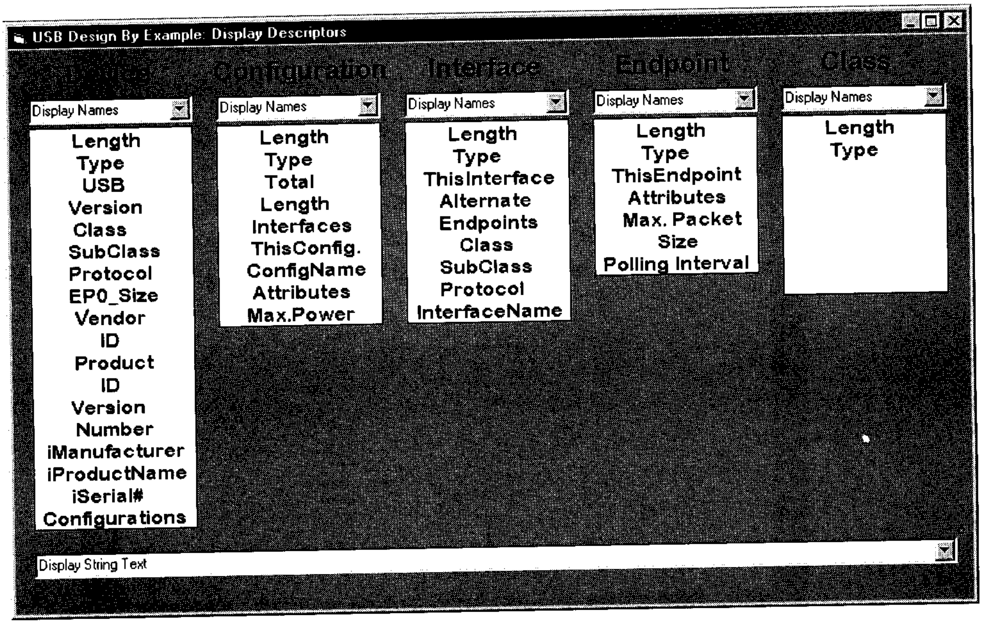


Figure 4-6. Descriptor display form

Example 1—Step 2: Initializing Program

As the program loads, Visual Basic sends a "loading form" event to the program. If a subroutine called FORM_LOAD is present, then this will be executed. Our example uses this subroutine to search for USB host controllers as shown in Figure 4-7. The example searches for up to four controllers—if you expect more, then add more "Host Controller" buttons. The buttons are declared as an array [HCD (3)], so no new software will be required for these additional buttons.

If a host controller is found, the program changes the background color of the corresponding button to green and enables that button. A system identifier for the host controller is stored in the button's TAG field for later use. If a host controller is not found, the program changes the background color of the corresponding button to orange and disables the button. The result of this FORM_LOAD subroutine is an indication of how many host controllers are present and a prompt to the user to select one to study.

```
' Look for Host Controllers.
' I limit the search to 3. There may be more but this is unlikely.
' The Host Controller Buttons are HCD(0) to HCD(3)
' Try opening the controller using its Symbolic Name
'
For ControllerIndex& = 0 To 3
   HostControllerName$ = "\\.\HCD" & ControllerIndex&
   HostControllerHandle& = CreateFile(HostControllerName$, &H40000000, 2, 0, 3, 0, 0)
   If HostControllerHandle& > 0 Then
      HCD(ControllerIndex&).Tag = HostControllerHandle&
      HCD(ControllerIndex&).BackColor = RGB(0, 256, 0) 'Green = GO
      HCD(ControllerIndex&).Enabled = True
      Else
      HCD(ControllerIndex&).BackColor = RGB(256, 128, 0) 'Amber = wait
      HCD(ControllerIndex&).Enabled = False
      End If
   Next ControllerIndex&
StatusBox.Text = "Select a Host Controller"
```

Figure 4-7. FORM_LOAD identifies host controllers

Example 1—Step 3: Choosing a Host Controller

This step does all of the USB-sensitive work in this program, so we'll move more slowly here. Clicking on a host controller button will start the USB device data collection process. Many operating system calls will be made, and I chose to encapsulate each of these in a separate subroutine. Having separate routines enables error-checking following each call, and the routine returns only if there are no errors. The approach allows for precise local error-checking, and the calling routine does not need to check for errors.

A Windows I/O subsystem consists of many nodes. These could be USB devices, SCSI devices, hard disk controllers, PCI bus connectors, or many others. The operating system refers to all I/O devices by their *node **detail** information* and their *node **connection** information*—this may appear verbose for our USB example, but remember that the I/O subsystem has to handle all I/O devices in a consistent way. Each I/O device family is connected to a **root node**, so our first task is to identify the USB root node.

Windows maintains symbolic names for all of the I/O devices it supports. We saw in "Example 1—Step 2: Initializing Program" that the root node name for a USB host controller was "HCDn." Using a standard system call, we can ask the operating system for the name of the node at the next level down in the I/O device hierarchy. We can repeat this process and thus identify all of the connections to all of the nodes. We are, in fact, traversing the tree structure of the I/O devices starting at the root.

Example 1—Step 3.1: Identifying the Root Hub Node

We have an operating system handle for the host controller (obtained in Form_Load and stored in the button's TAG field)—now we can request the name of the root hub using the DeviceIoControl system call. The operating system requires a two-stage process to get this name (Figure 4-8). The first system call returns the LENGTH of the name and the second returns the name in UNICODE format. The GetNameOf$ subroutine also converts the Unicode format into Visual Basic string format.

```
Function GetNameOf$(DeviceName$, DeviceHandle&, API_ID&)
Dim NameBuffer As UNameType
'
' First need to get the length of the name string
Status& = DeviceIoControl(DeviceHandle&, API_ID&, 0, 0, NameBuffer.Length, 260, BytesReturned&, 0)
If Status& = 0 Then ErrorExit ("Could not get LENGTH of " & DeviceName$ & " Name")
If NameBuffer.Length > 256 Then ErrorExit (Name$ & " Name > 256 Characters")
'
' . . . and then the string. It will be returned in UNICODE format
Status& = DeviceIoControl(DeviceHandle&, API_ID&, NameBuffer.Length, NameBuffer.Length, _
NameBuffer.Length, NameBuffer.Length, BytesReturned&, 0)
If Status& = 0 Then ErrorExit ("Could not get TEXT of " & DeviceName$ & " Name")
temp$ = "": i = 0  'A simple unicode to basic string conversion
Do While NameBuffer.UnicodeName(i) <> 0
temp$ = temp$ & Chr(NameBuffer.UnicodeName(i)): i = i + 2: Loop
GetNameOf$ = temp$
End Function

' Get the name of the host controller
HostController$ = GetNameOf("Host Controller", HCD(Index).Tag, &H220424)
StatusBox.Text = "Host Controller: " & HostController$ & " selected"
'
' Get the name of the Root Hub and open a connection to it
RootHubName$ = GetNameOf("Root Hub", HCD(Index).Tag, &H220408)
```

Figure 4-8. Getting the root hub name

We can now make a connection to the root hub using a CreateFile system call. This operating system call returns a handle we can use to discover information about this node. We'll use a DeviceIoControl system call to retrieve this information, and Figure 4-9 shows the node information returned. Important for our traversal of the device tree is RootHubNode.NodeDescriptor.PortCount, because this defines the number of connection points for other nodes.

Because we may want to get more information on this connection later, we store key parameters in a large data table. Note that we could not use the Listbox.ItemData feature because there is more than a single variable.

```
Public Type HubDescriptor
   Length As Byte:     HubType As Byte:   PortCount As Byte:    Characteristics(1) As Byte
   PowerOn2Good As Byte: MaxCurrent As Byte: PowerMask(63) As Byte: End Type

Public Type NodeInformation
   NodeType As Long: NodeDescriptor As HubDescriptor: HubIsBusPowered As Byte: End Type
```

Figure 4-9. Node information data structure

Example 1—Step 3.2: Probing Root Hub Connections

Now that we have identified a set of ports to the root hub, we request NodeConnectionInformation from the operating system so we can identify what, if anything, is connected to each port of the hub. A DeviceIoControl system call is used to retrieve this information, and Figure 4-10 shows the connection information returned. Information relevant for our traversal of the device tree is USBDeviceInfo.ConnectionStatus and USBDeviceInfo.DeviceIsHub. We use this information to decide if we need to reiterate on the ports of another hub.

```
Public Type DeviceDescriptor
   Length As Byte:  DescriptorType As Byte:  USBSpec(1) As Byte:  Class As Byte
   SubClass As Byte: Protocol As Byte:       MaxEP0Size As Byte:  VendorID(1) As Byte
   ProductID(1) As Byte:    DeviceRevision(1) As Byte:    ManufacturerStringIndex As Byte
   ProductStringIndex As Byte: SerialNumberStringIndex As Byte: ConfigurationCount As Byte: End Type

Public Type NodeConnectionInformation
   ConnectionIndex As Long: ThisDevice As DeviceDescriptor: CurrentConfiguration As Byte
   LowSpeed As Byte: DeviceIsHub As Byte: DeviceAddress(1) As Byte: OpenEndPoints(3) As Byte
   ThisConnectionStatus(3) As Byte: MyEndPoints(29) As EndPointDescriptor: End Type
```

Figure 4-10. Connection information data structure

Again, key information is logged into our USBDeviceInformation data table, and the device display is updated. Figure 4-11 shows the subroutine that implements the node traversal and data collection. This GetPortData function calls itself if it discovers that a hub is connected to one of the ports. No special programming is required in Visual Basic for this recursive operation, because all subroutines and functions automatically support recursion.

The GetPortData function will eventually complete when all branches of the tree have been traversed.

```
Function GetPortData&(Handle&, PortCount As Byte, HubDepth&)
Dim ThisDevice As Byte

For PortIndex& = 1 To PortCount
  Call GetNodeConnectionData(Handle&, PortIndex&)

  ThisDevice = 0 'default value, no device connected
  PortStatus& = DeviceData(DataIndex).ConnectionData.ThisConnectionStatus(0) 'save some typing!
  If PortStatus& = 1 Then
    ThisDevice = DeviceData(DataIndex).ConnectionData.DeviceAddress(0)
    DeviceData(DataIndex).DeviceHandle = Handle&
    End If
  'Create an indented display so that Hubs and their connections are easily seen
  Indent$ = " ": For i& = 1 To HubDepth&: Indent$ = Indent$ & ".": Next i&
  DeviceName$ = ThreeDecimalCharacters$(ThisDevice) & Indent$ & "    Port["
  Mid$(DeviceName$, 10) = ":"

  If PortStatus& <> 1 Then 'There is not a valid device on this port, tell user
    Device_Display.AddItem DeviceName$ & PortIndex & "] = " & ConnectionStatus$(PortStatus&)
    Else 'have a Device or a Hub connected to this port

    If DeviceData(DataIndex).ConnectionData.DeviceIsHub Then
'
' Need to discover how many ports are supported on this hub.
' Follow the same proceedure as we did for the root hub = get it's name, "open" it and get the node info.
      ExternalHubName$ = GetExternalHubName(PortIndex&, Handle&)
      ExternalHubHandle& = OpenConnection(ExternalHubName$)
      Call GetNodeInformation(ExternalHubHandle&)
      DeviceData(DataIndex).DeviceType = 2 'Hub
' LAST thing we do is update the display status of this device connection
      Device_Display.AddItem DeviceName$ & PortIndex & "] = Hub Connected"
'
' Discover what, if anything, is connected to the ports of this Root Hub
      Level& = GetPortData&(ExternalHubHandle&, _
DeviceData(DataIndex - 1).NodeData.NodeDescriptor.PortCount, HubDepth& + 1)

    Else 'we have a device connected to this port
      DeviceData(DataIndex).DeviceType = 3 'IODevice
      Device_Display.AddItem DeviceName$ & PortIndex & "] = IO Device Connected"
      End If 'USBDeviceInfo.DeviceIsHub
    End If 'PortStatus& <> 1
  Next PortIndex&
End Function
```

Figure 4-11. GetPortData function collects device data

Example 1—Step 4: Descriptor Display

The result of Step 3 is a completed USBDeviceInformation data table and a completed Device_Display. If the device list is longer than the display space allocated, then Visual Basic automatically creates scroll bars so that all of the device entries can be viewed.

The user selects a device by clicking on an entry in the Device_Display ListBox. The Device_Display_Click subroutine identifies the selected entry and passes control to the Display_Descriptor module.

All of the data we have accessed up to now has been provided from operating system tables. We now need to collect the descriptor information for the selected device, and this will involve sending USB requests to the selected device.

Example 1—Step 4.1: Collecting Device Descriptor Information

Request Packets (Figure 4-12) must be created and sent to the USB device. The same process is used to send any type of request: A packet is preformatted and a DeviceIoControl system call is made. To retrieve a Configuration Descriptor, for example, two calls must be made: The first retrieves just the 18-byte configuration descriptor, and the second uses the TotalLength parameter to read all of the descriptors. If the selected device has multiple configurations, then the data for each must be retrieved. Figure 4-12 shows the subroutine that implements this data collection.

```
Private Sub CollectDescriptors(Selected&)
'Collect all of the descriptors from the selected device and store them in the DescriptorData byte array
'Start with the Device Descriptor
For i& = 1 To 18: DescriptorData(i&) = DeviceData(Selected&).ConnectionData.ThisDevice.Contents(i& - 1): Next i&
Nexti& = 18
'Now get local copies of some key variables
Dim Configuration As Byte: Dim StringIndex As Byte
Handle& = DeviceData(Selected&).DeviceHandle
ConnectionIndex& = DeviceData(Selected&).ConnectionData.ConnectionIndex
ConfigurationCount = DeviceData(Selected).ConnectionData.ThisDevice.Contents(17)
For Configuration = 1 To ConfigurationCount
   TotalLength& = GetConfigurationDescriptor(Handle&, ConnectionIndex&, Configuration - 1)
'Copy the Configuration Descriptor into the DescriptorData byte array
For i& = 1 To TotalLength&: DescriptorData(Nexti& + i&) = PCHostRequest.ConfigurationDescriptor(i& - 1): Next i&
   Nexti& = Nexti& + TotalLength&: Next Configuration
'Check for Strings
StringIndex = 0
Do While TotalLength& <> 0
   TotalLength = GetStringDescriptor(Handle&, ConnectionIndex&, StringIndex)
   StringIndex = StringIndex + 1
   For i& = 1 To TotalLength&: DescriptorData(Nexti& + i&) = PCHostRequest.ConfigurationDescriptor(i& - 1): Next i&
   Nexti& = Nexti& + TotalLength&: Loop
End Sub
```

Figure 4-12. Retrieving all of the descriptor data

Example 1—Step 4.2: Interpreting the Configuration Descriptor

Step 4.1 returns a large, unformatted buffer of bytes that is a concatenation of the device descriptors. Rather than just display this raw data, the ParseDescriptor subroutine (Figure 4-13) interprets the data and extracts indexes into the buffer; these indexes are stored in the CHOICE ListBox.ItemData entries for easier display. The format of the descriptor data, with an initial LENGTH byte and a subsequent TYPE byte, makes the data parsing quite straightforward.

```
Private Sub ParseDescriptors()
DeviceCount = 0: ConfigurationCount = 0: InterfaceCount = 0: EndpointCount = 0: ClassCount = 0
Index& = 1
Do While DescriptorData(Index&) <> 0
  Select Case DescriptorData(Index& + 1) ' What TYPE of descriptor is this?
  Case 1
  DeviceCount = DeviceCount + 1: Choice(0).AddItem "Display Values"
  Choice(0).ItemData(Choice(0).ListCount - 1) = Index&
  Case 2
  ConfigurationCount = ConfigurationCount + 1: Choice(1).AddItem "Configuration " & ConfigurationCount
  Choice(1).ItemData(Choice(1).ListCount - 1) = Index&
  Case 3
  If StringCount <> 0 Then Choice(5).AddItem "String " & TwoHexCharacters$(CByte(StringCount)) & _
                                                        " = " & GetString$(Index&)
  StringCount = StringCount + 1
  Case 4
  InterfaceCount = InterfaceCount + 1
  Choice(2).AddItem "Interface " & ConfigurationCount & ":" & InterfaceCount
  Choice(2).ItemData(Choice(2).ListCount - 1) = Index&
  Case 5
  EndpointCount = EndpointCount + 1
  Choice(3).AddItem "Endpoint " & ConfigurationCount & ":" & EndpointCount
  Choice(3).ItemData(Choice(3).ListCount - 1) = Index&
  Case Else ' Must be a Class Descriptor
  ClassCount = ClassCount + 1
  Choice(4).AddItem "Class(" & TwoHexCharacters$(CByte(DescriptorData(Index& + 1))) & ") " _
                                                        & ConfigurationCount & ":" & ClassCount
  Choice(4).ItemData(Choice(4).ListCount - 1) = Index&
  End Select
  Index& = Index& + DescriptorData(Index&)
  Loop
'Fill out the default data for the Descriptors
For i% = 0 To 4
  If Choice(i%).ListCount = 1 Then 'this descriptor type is not present so remove it from the display
    Choice(i%).Visible = False: Descriptor(i%).Visible = False
    Else 'fill with data
    Choice(i%).ItemData(0) = Choice(i%).ItemData(1): Call AddDescriptorData(i%, 0)
    End If
  Next i%
If StringCount = 0 Then Choice(5).Visible = False
End Sub
```

Figure 4-13. Parsing the device descriptor information

Example 1—Step 4.3: Displaying Individual Descriptors

The descriptor display initializes with all text entries in the display boxes. The individual descriptors are listed in the CHOICE box above each display box as previewed in Figure 4-6. Individual text entries can be replaced by their value by clicking on an entry, or the whole descriptor data can be displayed by choosing its entry from a CHOICE box. Figure 4-14 shows a typical display following several user interactions.

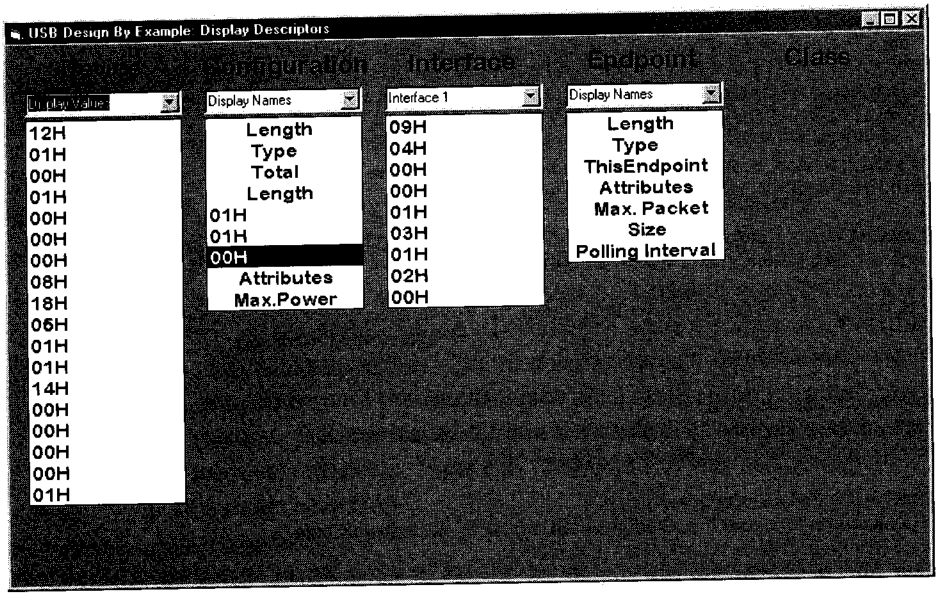


Figure 4-14. Descriptor display in operation

Example 1—Summary

The example demonstrated the code required to identify all of the USB Devices attached to a PC host. A specific device was identified. The example listed all devices, and a user selected one. Other programs could search through the collected information and choose a device based on some identifying criteria. Note that the device descriptor, which contains VendorID, ProductID, and Version#, is contained within the NodeConnectionInformation data structure, so searching on these parameters is straightforward. A targeted request packet was sent to the identified device. The example requested all of the device descriptors, but your program could send any valid request—the program structure is the same. "Low-level" passing of request packets was successfully demonstrated.

EXAMPLE 2: HID DISPLAY

The second example is simpler because we need to extract and display only a single system table. Windows will keep all HIDs in a single table regardless of their source (multiple USB controllers, PCI bus, or legacy ports). We will also extract some device-specific information because our application program in Chapter 6 will need to search on this.

We'll follow the same steps as in the first example, but there is less work to do.

Example 2—Step 1: Human Interface Design

This example is a display-only program, so we don't need buttons, etc. The value of this program is the data collected; this is displayed in a window (Figure 4-15). For each HID discovered, the program lists, if present, the VendorID and Manufacturer's name and the ProductID and Product name.

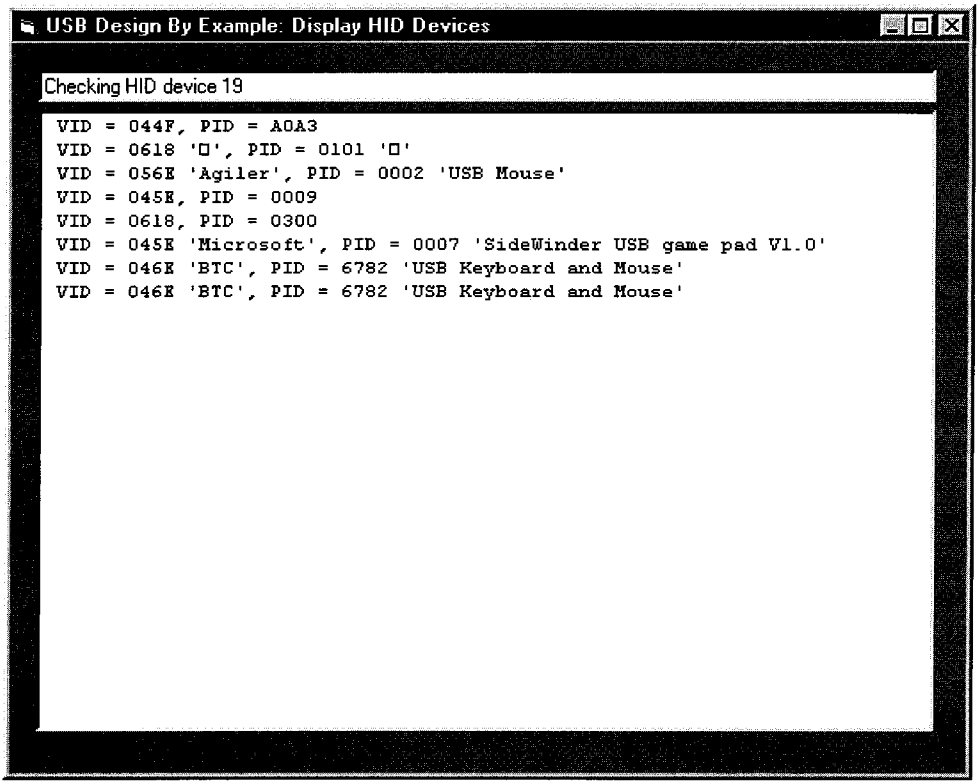


Figure 4-15. The first, and only, form

Example 2—Step 2: Initializing Program

Because there is no user interaction in this program, all of the work can be done in the Form_Load subroutine. We start searching for information at a system root node. The currently active I/O devices are listed at the system Plug and Play node, so we open a connection to this node, selecting information on HIDs (Figure 4-16).

```
Private Sub Form_Load()
'  Look for HIDs.
'
Dim hidGuid As Guid
'
'  First, get my class identifier
Call HidD_GetHidGuid(hidGuid.Data(0))
'
'  Get a handle for the Plug and Play node, request currently active HIDs
PnPHandle& = SetupDiGetClassDevs(hidGuid.Data(0), 0, 0, &H12)
```

Figure 4-16. Connecting to the Plug and Play node

The operating system will return a list of devices with as many entries as there are active devices. We must search from the beginning of the list looking for valid entries. Once a valid entry is identified, we can request the system name relating to this entry. A two-stage process is required to get the system name: The first call returns the length of the name, and the second returns a byte array containing the name. This byte array must be converted to a Visual Basic string so that a connection can be opened to the HID. Figure 4-17 shows the code required to identify and open an HID.

```
'  Let's look for a maximum of 20 HIDs
For HIDdevice& = 0 To 19
DeviceInterfaceData.cbsize = 28 'Length of data structure in bytes
'
'  Is there an HID at this table entry
'  If SetupDiEnumDeviceInterfaces(PnPHandle&, 0, hidGuid.Data(0), HIDdevice&, _
        DeviceInterfaceData.cbsize) Then
'  There is a device here, find out its system name.  Get the length of the name first
    RequiredLength& = 0
    success = SetupDiGetDeviceInterfaceDetail(PnPHandle&, DeviceInterfaceData.cbsize, _
        0, 0, RequiredLength&, 0)
'  Now get the name itself
    PredictedLength& = RequiredLength&
    FunctionClassDeviceData.cbsize = 5
    success = SetupDiGetDeviceInterfaceDetail(PnPHandle&, DeviceInterfaceData.cbsize, _
        AddressFor(FunctionClassDeviceData.cbsize), PredictedLength&, ReturnedLength&, 0)
'Convert data array to Visual Basic String
    Temp$ = "": i& = 0: Do While FunctionClassDeviceData.DataPath(i&) <> 0
    Temp$ = Temp$ & Chr(FunctionClassDeviceData.DataPath(i&)): i& = i& + 1: Loop
'Can now open this HID
    HidHandle& = CreateFile(Temp$, &HC0000000, 3, 0, 3, 0, 0)
```

Figure 4-17. Opening a connection to an HID

Example 2—Step 3: Displaying HID Information

Once we have a handle to a human interface device, getting information is straightforward because Microsoft has done all of the work already. The HID.DLL program contains subroutines to access the HID information, so all we have to do is make a library call and the desired information is placed in a buffer for us! Figure 4-18 lists the available function calls in HID.DLL that we can use. The first half of the list deals with Descriptor information, and the second half deals with the HID Report mechanism that is presented in the next section.

HidD_GetAttributes - returns VendorID, ProductID, Version#
HidD_GetManufacturerString
HidD_GetProductString
HidD_GetIndexedString
HidD_GetSerialNumberString
HidD_GetConfiguration
HidD_SetConfiguration
HidD_GetPreparsedData
HidD_FreePreparsedData
HidD_GetFeature
HidD_SetFeature
HidD_GetNumInputBuffers
HidD_SetNumInputBuffers
HidD_GetPhysicalDescriptor

Figure 4-18. HID.DLL is used to access HID data

Example 2—Summary

Working with human interface devices is simple because there is a lot of support within the operating system (I used Windows 98 in my examples, but other operating systems have similar capabilities). Our HID display example was easy because of the HID support within the operating system. Windows 98 supports the keyboard and mouse as HIDs, and the HID.DLL library provides access to the same software drivers.

Exchanging Data with an HID

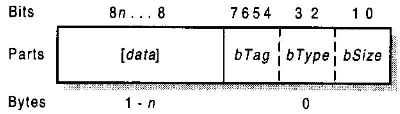
The mechanism used by the PC host to exchange data with an HID has been standardized via **reports**. A great deal of collaborative industry effort was used to define a mechanism that was both general enough to support every kind of human interface device available today, yet compact enough to be readily supported by low-cost I/O devices. The result was a form of pseudocode that is easily parsed by the operating system. The next few pages give an overview of the general solution and a specific solution that will be used in our Chapter 6 examples.

Report Descriptor

The primary and underlying goals of the report descriptor are:

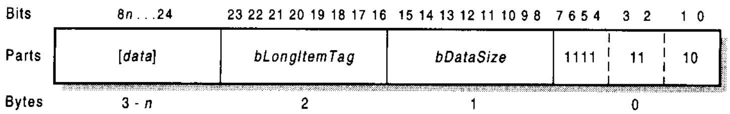
- Be as compact as possible to save device data space.
- Allow the software application to skip unknown information.
- Be extensible and robust.
- Support nesting and collections.
- Be self-describing to allow generic software applications.

Report descriptors are composed of pieces of information called **items**. An item is a piece of information about the device. All items have a one-byte prefix that contains the item tag, item type, and item size as shown in the upper section of Figure 4-19. An item can include optional data. The size of the data portion of an item is determined by its fundamental type. There are two basic types of items: short items and long items. A short item's optional data size may be 0, 1, 2, or 4 bytes. A long item's *bSize* value is always 2. The lower section of Figure 4-19 illustrates possible values within the one-byte prefix for a long item. All examples in this book use short items.



Part	Description
bSize	Numeric expression specifying size of data: 0 = 0 bytes 1 = 1 byte 2 = 2 bytes 3 = 4 bytes
bType	Numeric expression identifying type of item where: 0 = Main 1 = Global 2 = Local 3 = Reserved
bTag	Numeric expression specifying the function of the item
[data]	Optional data

Generic Format



Long Item Format

Figure 4-19. Format of an item within an HID report

The report descriptor is unlike other descriptors in that it is not simply a table of values. The length and content of a report descriptor vary depending on the number of data fields required for the device's report or reports. The report descriptor is made up of items that provide information about the device. The first part of an item contains three fields: item type, item tag, and item size. Together these fields identify the kind of information the item provides.

A report descriptor **must** include each of the following to describe data about a control (all other items are optional):

- Usage Page
- Usage
- Main Items
- Logical Minimum
- Logical Maximum
- Report Size
- Report Count

These items are organized into a tree structure as shown in Figure 4-20.

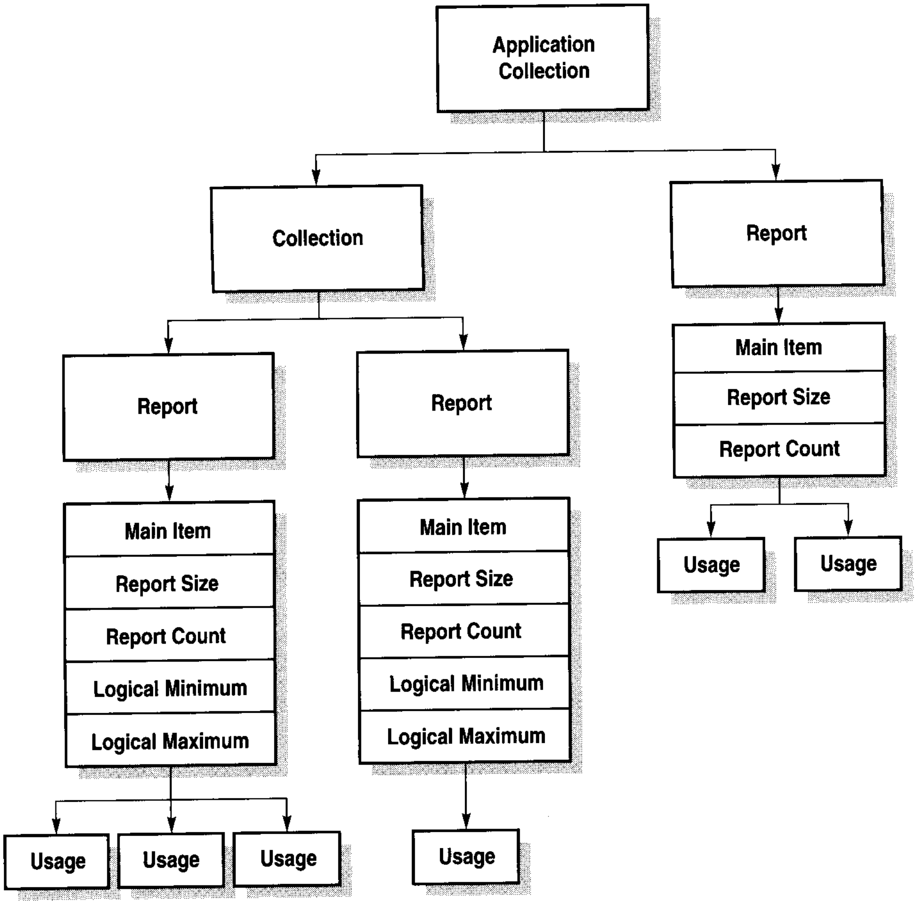


Figure 4-20. The structure of a report descriptor

Figure 4-21 shows an example of a three-button mouse. Remember that this I/O device can report any of three button pushes and changes in its X-Y coordinate position. No information is output to a mouse.

```
Usage Page (Generic Desktop),                 ;Use the Generic Desktop Usage Page
Usage (Mouse),
    Collection (Application),                 ;Start Mouse collection
    Usage (Pointer),
    Collection (Physical),                    ;Start Pointer collection
    Usage Page (Buttons)
    Usage Minimum (1),
    Usage Maximum (3),
    Logical Minimum (0),
    Logical Maximum (1),                      ;Fields return values from 0 to 1
    Report Count (3),
    Report Size (1),                          ;Create three 1-bit button fields
    Input (Data, Variable, Absolute),         ;Add fields to the input report
    Report Count (1),
    Report Size (5),                          ;Create 5-bit constant field
    Input (Constant),                         ;Add field to the input report
    Usage Page (Generic Desktop),
    Usage (X),
    Usage (Y),
    Logical Minimum (-127),
    Logical Maximum (127),                    ;Fields return values from -127 to 127
    Report Size (8),
    Report Count (2),                         ;Create two fields (X & Y position)
    Input (Data, Variable, Relative),         ;Add fields to the input report
End Collection,
End Collection
```

Figure 4-21. Report descriptor of a 3-button mouse

Examination of Figure 4-21 shows how each of the items is used. An array of bytes is required for the I/O device, and a tool called the HID descriptor tool (to be described in Chapter 5) can be used to generate the text of Figure 4-21 and the data structure that will be embedded in the I/O device.

Figure 4-22 shows the report descriptor for a standard keyboard. An output report is required to light the NumLock, CapsLock, and ScrollLock LEDs, and the input report allows for multiple keys to be read at a time.

```
Usage Page (Generic Desktop),                ;Use the Generic Desktop Usage Page
Usage (Keyboard),
        Collection (Application),            ;Start Keyboard collection
        Usage Page (Key Codes),
        Usage Minimum (234),
        Usage Maximum (231),
        Logical Minimum (0),
        Logical Maximum (1),                 ;Fields return values from 0 to 1
        Report Count (8),
        Report Size (1),
        Input (Data, Variable, Absolute),    ;Add fields to the input report.
        Report Count (5),                    ;LED Report
        Report Size (1),                     ;Create 8-bit constant field
        Usage Page (LEDs),
        Usage Minimum (1),
        Usage Maximum (5),
        Output (Data, Variable, Absolute),
        Report Count (1),
        Report Size (3),
        Output (Constant),                   ; 3 bits of Padding
        Report Count (6),                    ; 6 characters are buffered
        Report Size (8),
        Logical Minimum (0),
        Logical Maximum (101),
        Usage Page (Key Codes),
        Usage Minimum (0),
        Usage Maximum (101),
        Input (Data, Array),                 ; 6-character buffer
        End Collection,
End Collection
```

Figure 4-22. Report descriptor of a keyboard

Design Example

The first few examples in Chapter 6 will exchange fixed length data buffers using the report descriptor shown in Figure 4-23. The contents of the buffers will be known by the PC host applications program and the I/O device firmware program—in USB specification terms, this means "vendor-specific."

```
Usage Page (Vendor Specific),                 ;The examples use unique data
Usage (Vendor Specific),
Collection (Application),                     ;Start our collection
        Usage Minimum (0),
        Usage Maximum (255),
        Logical Minimum (0),
        Logical Maximum (255),                ;Fields can be any byte value
        Report Count (8),                     ;8 bit fields = 1 byte
        Report Size (1),                      ;Byte count
        Input (Data, Variable, Absolute),     ;Add fields to the input report.
        Output (Data, Variable, Absolute),    ; Output report is the same size
        End Collection,
End Collection
```

Figure 4-23. An example report descriptor

Two example subroutines will be written—**WriteUSBdevice** and **ReadUSBdevice**. The PC host will create a buffer and pass it to the WriteUSBdevice routine, which will create a report and use HID.DLL to deliver the report to the target I/O device. Similarly, ReadUSBdevice will create a GetReport request, and HID.DLL will retrieve a buffer from the target I/O device. The source for the WriteUSBdevice and ReadUSBdevice routines is included in the Chapter 4 directory on the CD-ROM.

CHAPTER SUMMARY

This chapter has demonstrated the key pieces of PC host software required to access USB devices. The examples were written in Visual Basic so that many of the complexities of Windows applications programming could be hidden and focus could be applied to the USB aspects of the design. If "low-level" access is required, the approach taken by Example 1 is appropriate. The "high-level" approach taken by Example 2 is simpler because most of the required software is supplied in an operating system library, and all we need to do is call the appropriate routines. The generalized report mechanism for exchanging data between a PC host and USB I/O device is verbose, but two subroutines, WriteUSBdevice and ReadUSBdevice, were provided to simplify the examples used in this book. The source code for all of these examples is provided on the CD-ROM, and the "Learning Version" of Visual Basic is adequate to modify the examples to suit your needs.

CHAPTER 5
DEVELOPMENT TOOLS

In Chapter 3 we learned that an I/O device consists of a USB transceiver, a Serial Interface Engine (SIE), a protocol controller (typically a microcontroller), and interfaces to the real world. The USB Specification defines SIE functionality, and all available USB devices are tested for compliance to that specification. Some vendors have augmented the base SIE capabilities to offload the protocol controller; this augmentation is also included within the USB Specification. This chapter assumes that we will not develop an SIE, but that instead we'll use one of the available products. Thus, the development task consists of engineering a protocol controller that interfaces to the SIE and to the real world.

The choice of protocol controllers is large, and the development task will be specific to the protocol controller chosen. This chapter starts at a high general level and then describes different groups of solutions using a representative example from each group.

The Development Environment

In this section the I/O device that we are developing is called the **target** and the PC platform is called a **host**—because it will host all of our development tools. Figure 5-1 shows an ideal development environment where the target includes its own USB-connected PC platform.

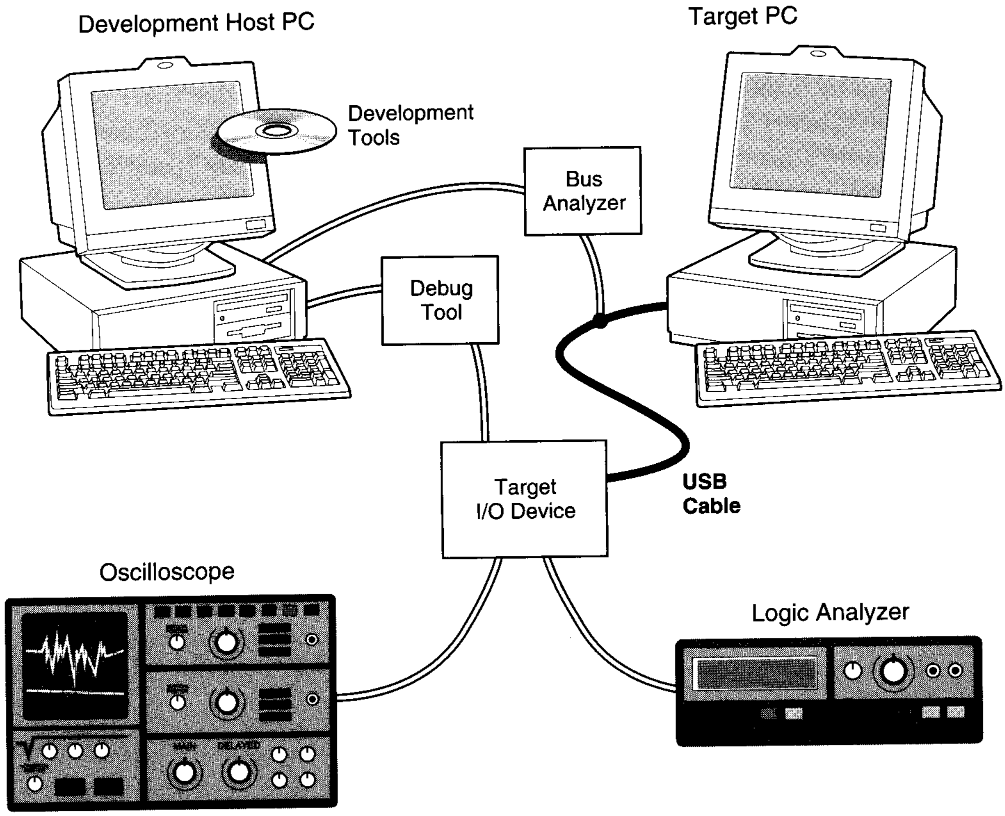


Figure 5-1. Ideal development environment

The logic analyzer and oscilloscope are used to capture signals on the target board. The bus analyzer captures all of the packets on the USB cable. The target PC runs the enumeration and then the application for our target board. Having a separate PC gives us flexibility and control during the debug process—if the target PC were to hang during development (a typical situation during code debugging), we still have control via the development host. The debug tool will vary with different groups of solutions, but its role of connecting the development PC host to the target is unchanged: Software developed on the development host can be downloaded into the target and its execution controlled and observed by the debug tool. Figure 5-2 shows the development environment that will be used for the examples in Chapter 6.

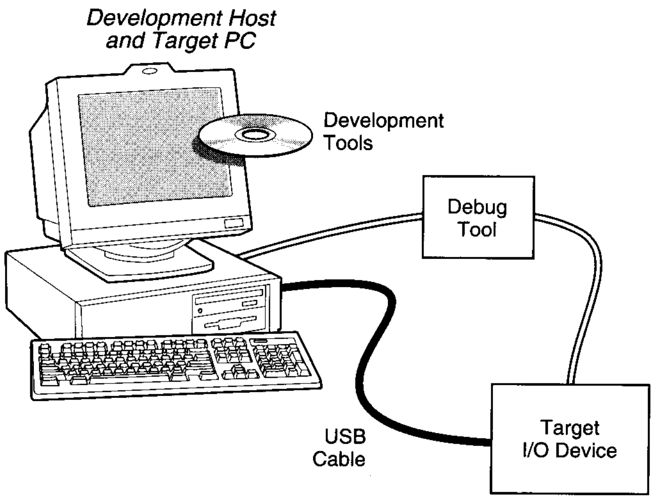


Figure 5-2. Adequate development environment for Chapter 6 examples

The examples in Chapter 6 start by being relatively easy; they are simple enough that neither a logic analyzer nor an oscilloscope is required. Only a limited number of USB packets will be used, so a bus analyzer is not essential. Finally, the risk of a system crash is small with our Chapter 6 examples because the examples are predebugged; thus, it is acceptable to use a single PC for both tasks. (To generate the examples, I used the equipment shown in Figure 5-1.) Once we move beyond the Chapter 6 examples, however, we'll need more development tools. First we'll add a bus analyzer (for a range of available equipment, see the Chapter 6/bus analyzer directory on the CD-ROM). The software environment is similar for all target implementations. Before discussing the software development and debug environment, let's look at the different development hardware that is available.

TARGET IMPLEMENTATIONS

The USB I/O device can be implemented in three different ways (Figure 5-3).

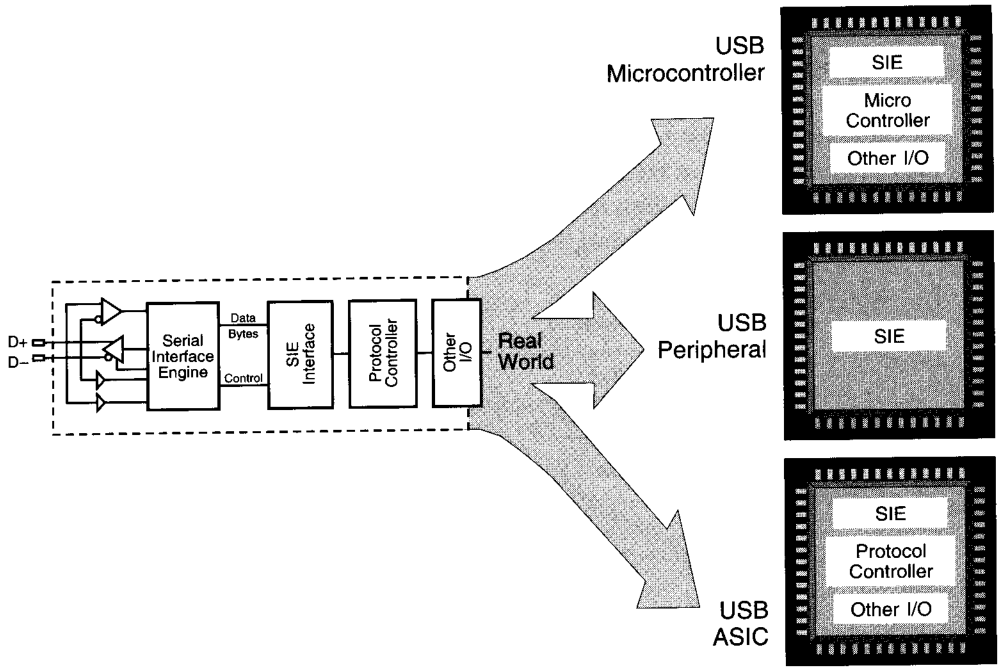


Figure 5-3. USB I/O device implementation options

The **USB microcontroller** integrates the SIE and a microcontroller onto a single component. The microcontroller contains data memory that can be expandable off-chip. The microcontroller can also contain program memory enabling a single chip solution to be implemented. Program memory can be expanded off-chip in some implementations. Two general development approaches are taken depending on the expansion capabilities of the microcontroller:

- The **"bond-out"** solution for single chip microcontrollers
- The **"debug monitor"** solution for external program memory microcontrollers

The **USB peripheral** contains the SIE and associated endpoint FIFOs in a single component. The protocol engine is implemented by an attached processor that could be any processor! The attached processor views the USB peripheral the same as other peripherals, such as a serial port or a network controller: The USB peripheral has status and control registers, and the peripheral is either the source

or recipient of the data. Two common connections to the attached processor are a three-wire I^2C bus and an 8-bit parallel bus if high performance is required. From a development perspective, this solution is developed and debugged using tools targeted for the attached processor.

The **USB ASIC** integrates the SIE, controller, and all required I/O into a single chip. This integrated chip results in the lowest product cost for the USB I/O device but the highest development cost. The development environment is similar to the bond-out solution except that different code generation tools are used on the development PC.

A Bond-out Example

A microcontroller with embedded program memory and only I/O signals on its pins is almost impossible to debug because of the limitations on observing its operation. To enable easy design and debug of these single-chip solutions, most manufacturers provide bond-out versions of their microcontrollers (Figure 5-4).

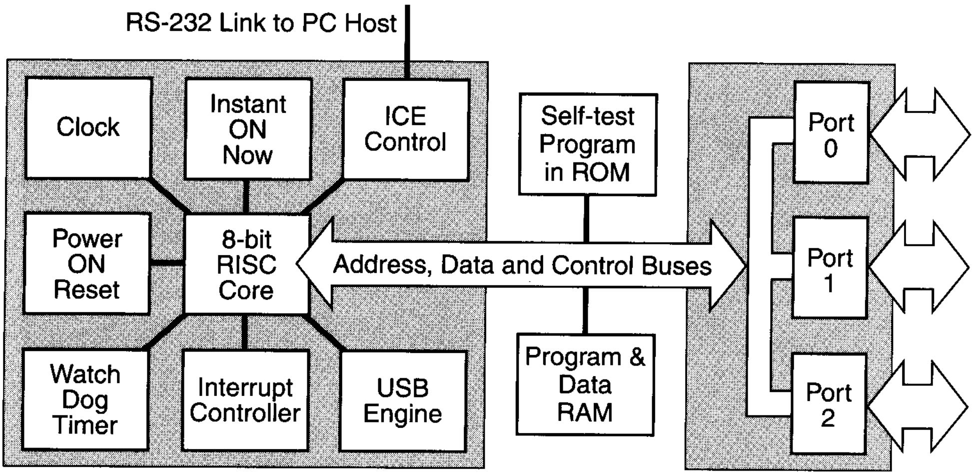
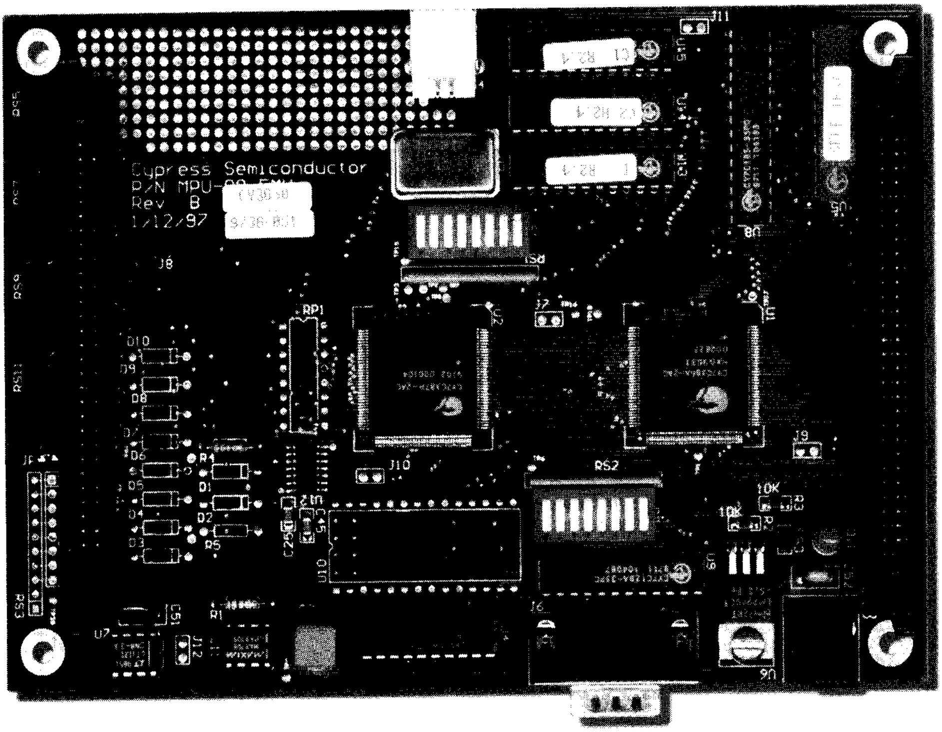


Figure 5-4. Bond-out part needs a bigger package

The development component is referred to as "bond-out" because extra internal signals not found on the pins of a production part are bonded-out to pins in a larger package. Providing these signals gives increased visibility into the operation of the microcontroller; special control circuitry can also be added to single-step the program, halt it at specific address or data accesses, and trace operation. During the debug phase, the program is stored in RAM memory because it is writable at execution speeds—we do not need to program and erase

EPROMs to edit the program. The added capabilities and large packages of a bond-out part mean that it is a lot more expensive than the production part, but we need only one.

Figure 5-5 shows a representative bond-out development kit. This unit from Cypress Semiconductor includes a cable that operates just like a production microcontroller as far as the target I/O board is concerned.



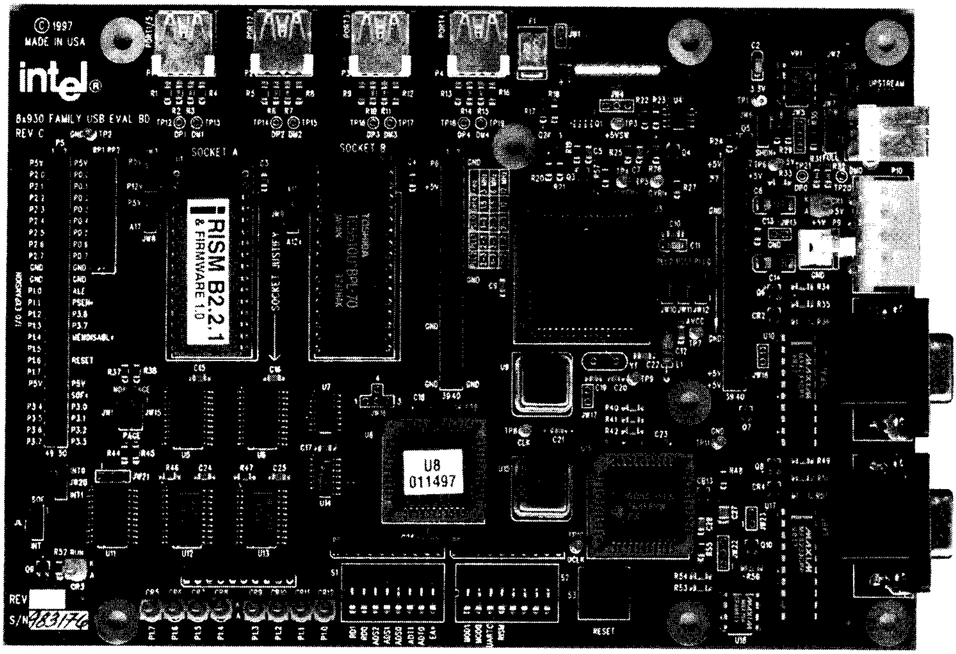
Courtesy of Cypress Semiconductor Corp.

Figure 5-5. Example of debug board using a bond-out component

A Debug Monitor Example

If the microcontroller executes from external program memory, it is straightforward to build a debug monitor development tool. The example in Figure 5-6 is an Intel 8x930 board that includes more program memory, both EPROM-based and RAM-based. Following power-up, the microcontroller executes from the on-board EPROM where a monitor program is stored. The monitor program is used to download a program via the serial port into RAM and to control its execution. The benefit of this approach is that there is no difference between the component used for development and the one used for production.

The 8x930 board supports a range of USB microcontrollers: The 8x931 has a fast MCC51 core and a large collection of specialized I/O devices; the 8x930 has a faster MCS251 core (enhanced MCS51 with larger address range, improved addressing modes, orthogonal instruction set that operates on bits, bytes, words and dwords). Each of these core microcontrollers is available as an I/O device or as a compound I/O device with a 4- or 6-port hub. User manuals for this product line are included on the CD-ROM.



Courtesy of Intel Corp.

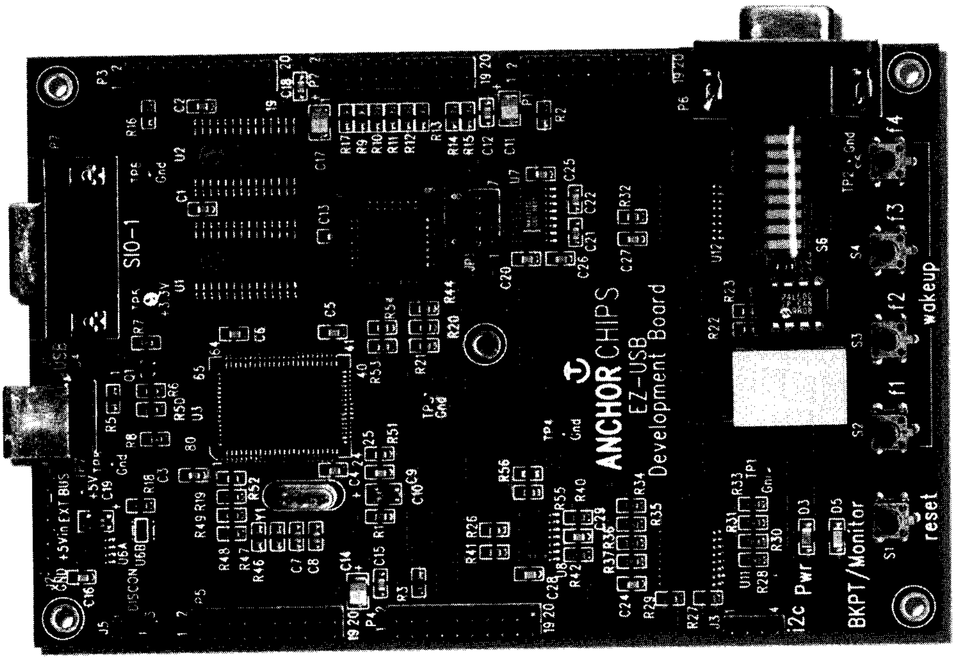
Figure 5-6. Example of debug monitor environment

An Integrated Debug Monitor Example

An innovative approach to the debug monitor solution has been taken by Anchor Chips with their EZ-USB product line.

The EZ-USB product is designed to download a program via the USB cable during enumeration. The development board in Figure 5-7 downloads a debug monitor during enumeration. To download the target program, you can use either a Windows application program or the debug monitor. The development environment is fully symbolic and includes a Keil C compiler as well as the assembler and linker toolkit.

The development board contains more hardware for test applications and debugging but uses exactly the same microcontroller device that will be used in production.



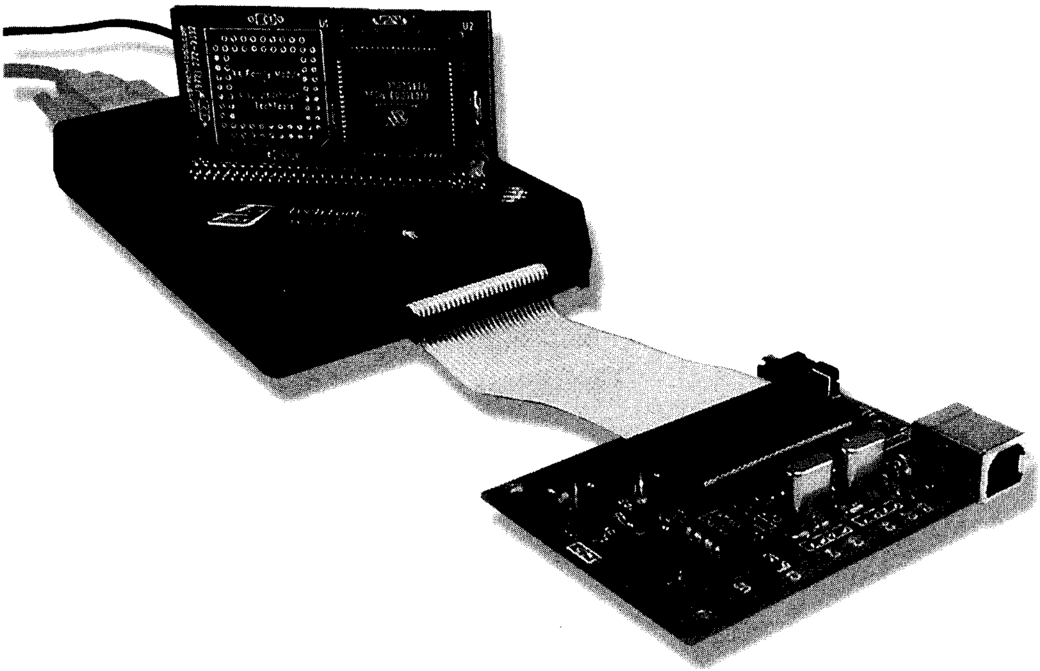
Courtesy of Anchor Chips, Inc.

Figure 5-7. EZ-USB has soft-loadable program

USB Peripheral Example

Figure 5-8 shows a representative development system for a USB peripheral.

This example is a Clearview Mathias system from TechTools designed to support the PIC family of microcontrollers. In this example the USB connection is provided by a Philips PDIUSBD11 I^2C-to-USB peripheral.



Courtesy of TechTools, Inc.

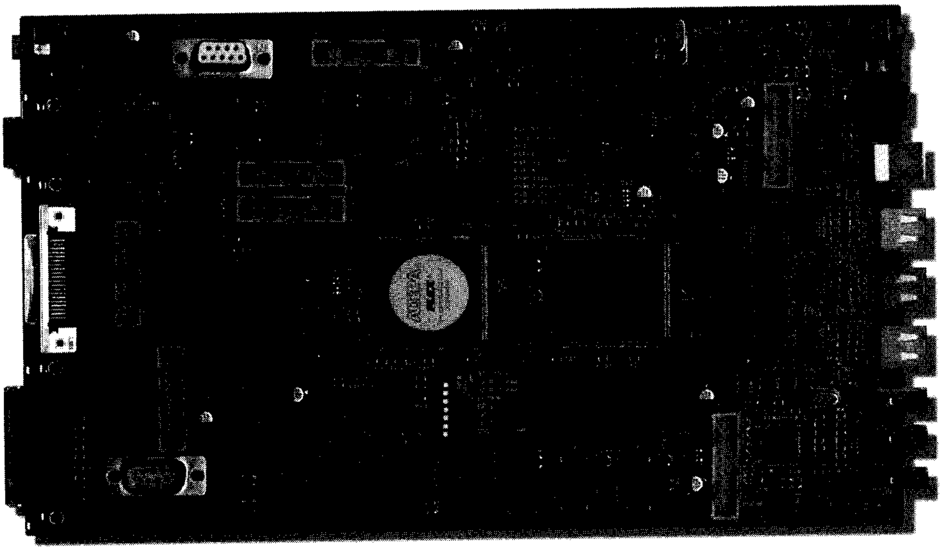
Figure 5-8. Example USB peripheral development system

The Clearview Mathias system uses bond-out components from MicroChip to provide the control, monitoring, and download capabilities required for design and debug. A cable from a personality module plugs into the I/O target system, and this cable operates just as a production microcontroller would operate.

USB ASIC Example

There is a lot more development work to do with the ASIC approach to USB I/O device design, because all the component hardware must also be created. However, this approach gives the designer much more freedom in the design, which can be tuned exactly to the requirements of the I/O device. The initial hardware design consists of configuring a "sea-of-gates" ASIC with the required elements of the design. VAutomation's solution has many prepackaged building blocks, such as the SIE, device and host controller, digital phase lock loop circuits, and a microcontroller interface. VAutomation also has a full 8-bit RISC microcontroller available as an ASIC library element to include in a design.

Figure 5-9 shows the development board, which includes hardware interfaces to audio, video, and parallel and serial ports. VAutomation includes example programs that exercise all of these interfaces.



Courtesy of VAutomation, Inc.

Figure 5-9. USB ASIC development board

Software Development Tools

Software for the protocol controller will be created on the development PC host. Tools specific for the microcontroller architecture will be used to write the code that delivers the responses to the PC host's requests and that implements the real-world I/O interactions. For most of the examples in this book, I have used assembler code for the microcontroller because there is a large amount of bit-handling and direct I/O manipulation. C compilers are also available for several microcontrollers; for a medium-to-large I/O design, you should first consider using C because the code will be easier to modify and maintain.

Most microcontroller vendors supply example code that implements the enumeration stage of a design. In the next chapter, we'll work through an example to see how the protocol controller interfaces to the Serial Interface Engine. Once you have written (or obtained) this enumeration code, you can reuse it on all projects, because the enumeration code does not change the personality of an I/O device; instead, such change is caused by entries in the descriptor tables. The run-time code will also change, of course, because different real-world interfaces will be used. In a USB I/O device project, you can take advantage of having a large opportunity to write **modules** of code—the microcontroller code behind an Interface Descriptor will be reusable on all designs that use the same descriptor.

Once the program code is written, it is assembled/compiled and an executable module is created. This module must be downloaded into the target system for debug. All of the development systems described in this chapter allow this module to be downloaded into RAM for the debug process—we don't have to program an EPROM yet!

A debug monitor program will already be resident in the target system—this will be PROM or can be downloaded before our target program. When creating the target program, ensure that it does not use any of the memory space or I/O resources of the debug monitor program.

The debug monitor program will be used to control execution of our target program. The debug monitor program also allows memory, microcontroller registers, and I/O ports to be displayed and modified. The target program can be single-stepped or run with breakpoints. The debug monitor sets a breakpoint by substituting a "trap" instruction in place of the actual instruction at the breakpoint address. When the microcontroller reaches this address, it switches from executing the target program back into executing the debug monitor program so that variables can be analyzed and program execution studied.

Some tools include a real-time trace facility that captures the execution address of the microcontroller for later analysis. This trace tells you exactly where the microcontroller has been. The trace can be stopped when the microcontroller accesses certain memory addresses, typically those it should not be addressing anyway, and the trace is studied to understand where the microcontroller "took a wrong turn." A trace tool, or attached logic analyzer with similar characteristics, is invaluable when you are trying to track down program bugs that send the microcontroller off-course.

In Chapter 6 we'll see that most of the protocol controller program runs under interrupt control. While this interrupt-driven approach is the most efficient method to use, it can create bugs that are hard to track down if we are not precise with interrupt handling. More care must be taken on microcontrollers that support multiple levels of interrupt nesting—a stack overflow is a common bug in such programs and is hard to track down.

In the Chapter 5 directory on CD-ROM, I have included a set of tools for the MCS 51 microcontroller. These tools were kindly provided by Keil Corporation, and they can be used to re-create all of the MCS 51 examples in the next chapter. These free tools are 100 percent functional but have been modified so they can create a target program only up to 2 KB long. This is ample for our examples. A set of tools that supports programs greater than 2 KB is available directly from Keil. The Keil tools on the CD-ROM also incorporate a sophisticated debug monitor program called Dscope that allows full symbolic access to all of the variables and memory locations—this is a lot easier than dealing with hexadecimal numbers.

USB-SPECIFIC TOOLS

To ease the development of the target I/O board, a variety of tools is available and included on the CD-ROM.

USB Single Step

The enumeration phase, when controlled by the operating system, is very fast. USB Single Step puts the USB host software into a diagnostic mode, which lets us single-step through each packet exchange as shown in Figure 5-10.

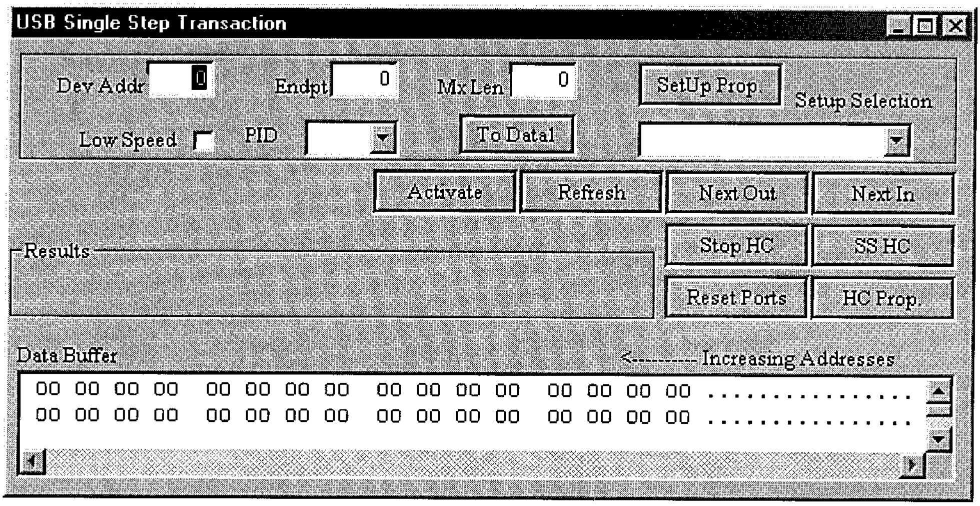


Figure 5-10. USB single-step tool

Host requests can be made in the same order as the enumeration phase, and the responses provided by the I/O device can be observed. Incorrect data or data lengths are easy to sift out at this stage. Once the I/O device successfully completes the enumeration phase, data packets can be sent and received to check the run-time operation.

USB View

USB View uses the data tables and USB routines of the operating system to draw a table of attached USB devices (Figure 5-11). The device descriptors are read and displayed.

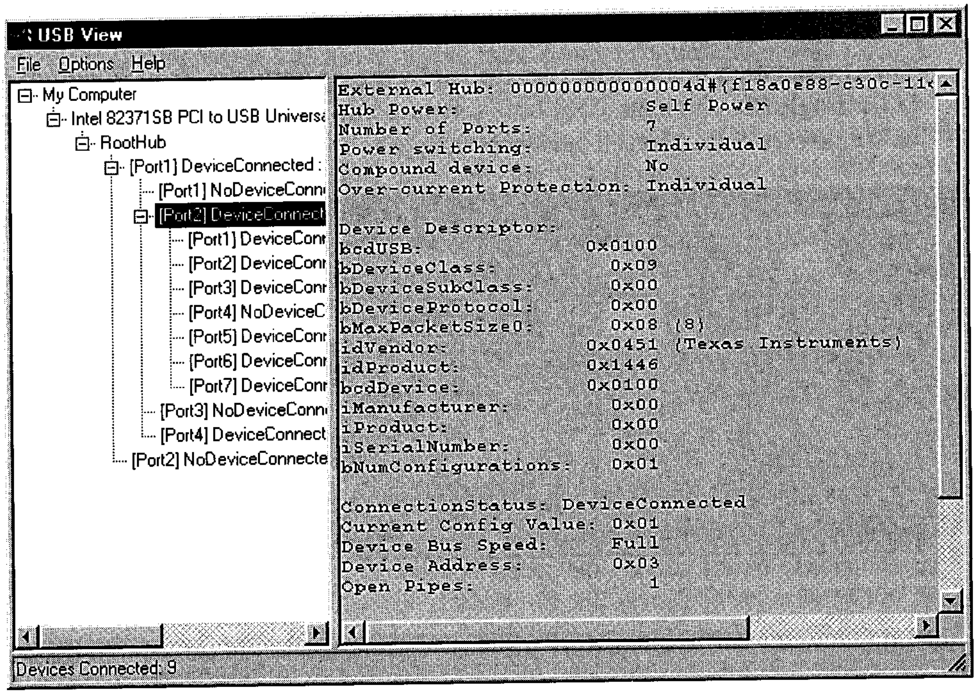


Figure 5-11. Operating system's view of USB devices

Hidview

Starting Hidview puts the operating system USB software into diagnostic mode for newly attached HID devices. The tool can then observe and monitor HID reports (Figure 5-12).

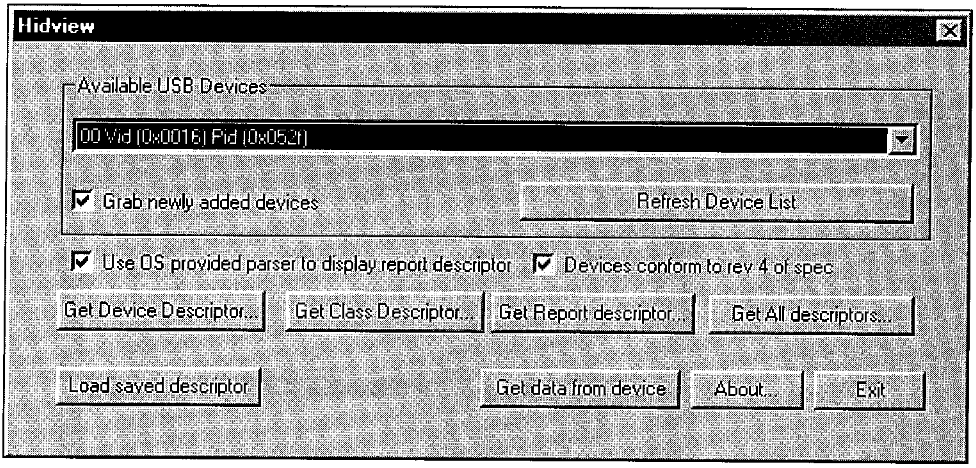


Figure 5-12. Debug of HID device using Hidview

HID Table Creator

An HID **report** is a complex, compacted data structure. The HID table creator is an interactive program that guides you through creating a report descriptor (Figure 5-13). You can save the output in multiple formats that can be pasted into an assembler file or used as a C declaration file. This tool saves a lot of time.

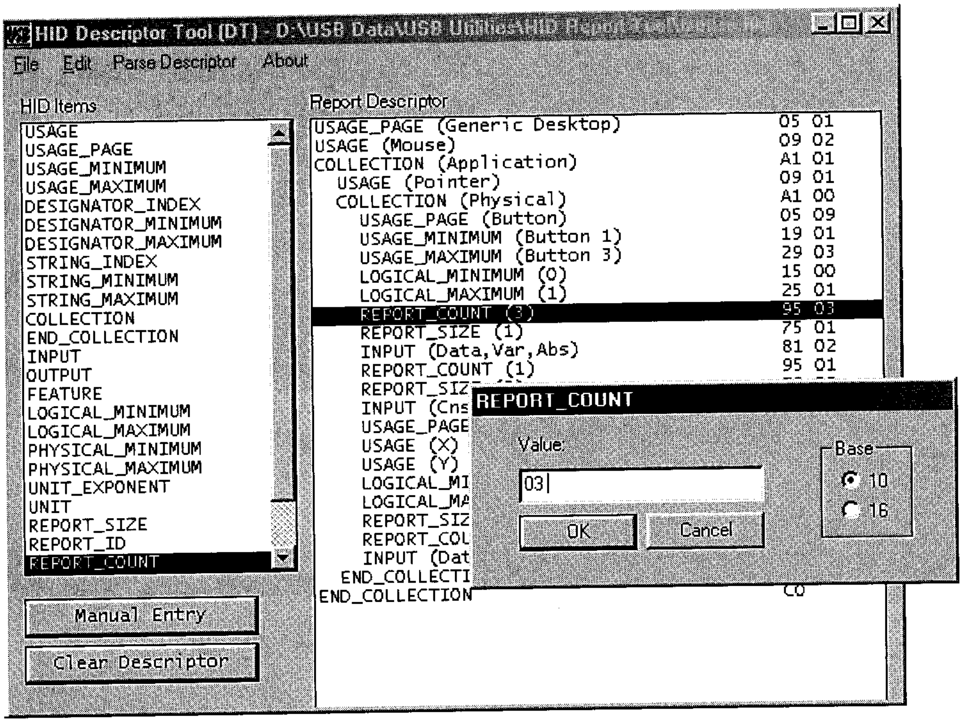


Figure 5-13. Use HID tool to create report descriptors

Chapter Summary

The development environment for a USB I/O device is mature. Different tools are available that match the target solution for the I/O device. These tools are available from multiple vendors and support their microcontroller solutions very well. Third-party tools are also available to support development efforts, and I have included a suite of these tools on the CD-ROM. I've also provided several host-based programs that will help in the development of your I/O device.

So what are we waiting for? Let's move on to the next chapter and build something!

CHAPTER 6
BUTTONS AND LIGHTS

In this chapter we'll build several I/O devices, using what we learned about PC host software in Chapter 4 and what we learned about the responsibilities of an I/O device in Chapters 2 and 3. To make it easier to focus on the method, we'll start with a simple design and gradually expand the design:

- **Example 1:** Simple design implements button functions and LEDs using a microcontroller that has an integrated USB port.
- **Example 2:** Simple design implements button functions and LEDs (same as above) using a generic different microcontroller and an external USB port.
- **Example 3:** Design includes adding more ports (independent of microcontroller).
- **Example 4:** Design includes adding lots more ports (independent of microcontroller) and supporting two configurations.

Figure 6-1 represents the overall design task. A Visual Basic application program on the PC host writes blocks of a buffer to an I/O device and reads buffers of data from an I/O device. The format of these data buffers will be known by the application program and by the microcontroller firmware.

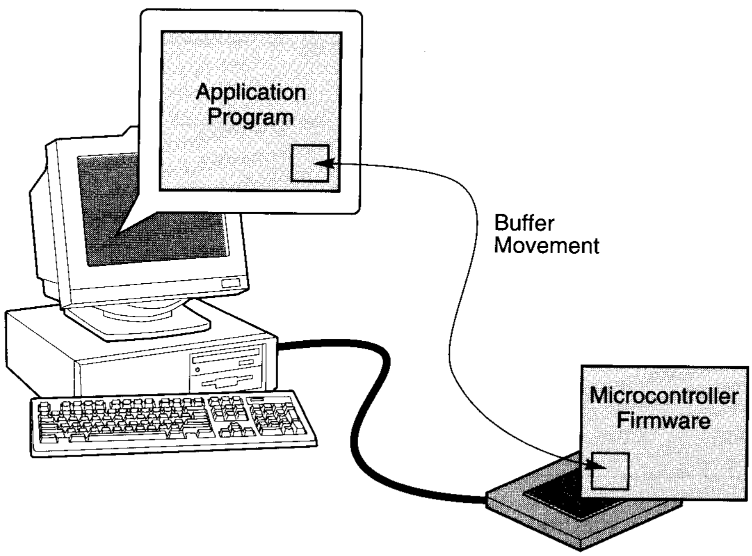


Figure 6-1. Overall task design

We'll have multiple examples, so each I/O device, its hardware, and its matching firmware must be uniquely identified by its Product Code and Name. The Visual Basic application program will first "open" a named I/O device to ensure that the hardware matches the software. The program will also "close" the connection as it exits.

When you develop an I/O device[1], you choose a USB interface, a microcontroller (or a microcontroller with a USB interface), and the required I/O devices (for guidance, see Appendix A). Then you write the subroutines that respond to the commands sent by the PC host.

EXAMPLE 1: SIMPLE DESIGN, INTEGRATED USB PORT

This simple example implements buttons and lights. In the example, the microcontroller has an integrated USB port.

The application goes through various stages of initialization:

- The I/O device is attached and enumerated. The I/O device enumerates as an HID device, and the system creates an entry for it in the HID device list attached to the root Plug and Play node. The second software example in Chapter 4 explained this operation in detail.
- The Visual Basic application starts executing. The Visual Basic application searches the list for "Buttons and Lights." If this is not present, the application prompts the user to plug in the I/O device. Because we haven't built our hardware yet, the Visual Basic program is going to wait a long time to detect the presence of an I/O device!

---

[1] An alternative method using synthesizable cores was discussed in Chapter 4. This chapter will focus on the lower volume implementation using a microcontroller.

Example 1—Step 1: Design the Hardware

Figure 6-2 shows our first USB I/O device: a simple, bus-powered I/O device consisting of a single 8-bit input port, with "buttons," and a single 8-bit output port, with "lights." I'm using a simple I/O device so we can focus on the **method.**

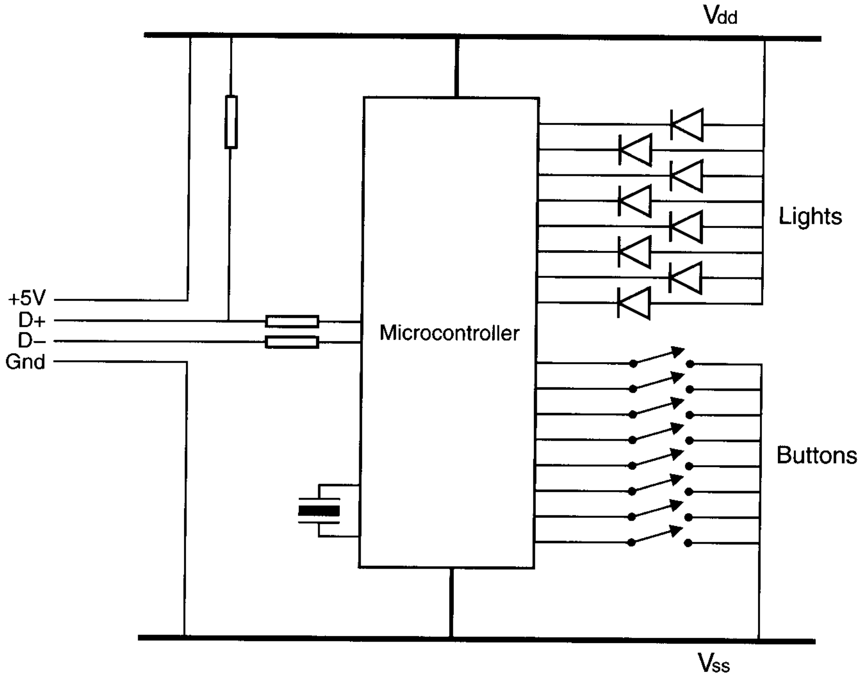


Figure 6-2. Our first USB I/O device

In this first example, design work consists mostly of writing firmware that is executed during the enumeration phase of initialization. The firmware looks quite verbose, but the good news is that we don't need to change it much for all of the other examples. Let's work through the example slowly so that we fully understand the steps.

Example 1—Step 2: Complete the Descriptors

Our first step is to complete the descriptors for this HID device. We have one interface and one configuration in this single device. The interface descriptor will point to an HID descriptor that in turn points to our single report. The report descriptor was generated using the HID tool introduced in Chapter 5. Figure 6-3 shows the descriptors for our first example.

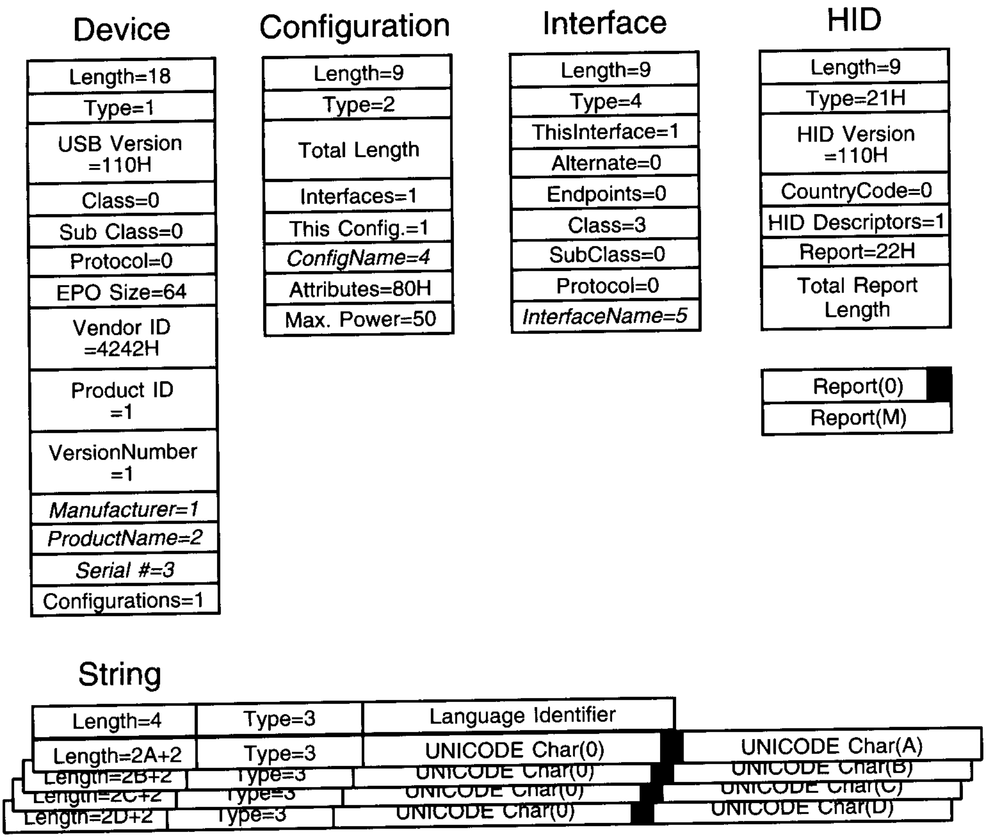


Figure 6-3. Descriptors for our first example

Table 6-1 lists the Standard Request Codes and the Class Request Codes that the PC host may send to our I/O device. Each request can be targeted at the device, an interface, or an endpoint, so the number of possible requests is quite large. Our general response to each request is also listed, but the details will vary. With so few features in this first example, many of the responses will be null subroutines. Our software structure allows for fuller responses in later examples.

Table 6-1. Requests that our I/O device must respond to

Type	Request	Our Action*
Standard	Get_Status	Return current status
Standard	Clear_Feature	Do nothing, we have no clearable features
Standard	Set_Feature	Do nothing, we have no settable features
Standard	Set_Address	Do it
Standard	Get_Descriptor	Return requested descriptor
Standard	Set_Descriptor	Do nothing, our descriptors are static
Standard	Get_Configuration	Return it or 0 if not configured
Standard	Set_Configuration	Do nothing, we have only one
Standard	Get_Interface	Return it
Standard	Set_Interface	Do nothing, we have only one
Standard	Sync_Frame	Report error since we are not an Async device
Standard	all others	Report error via a Stall
Class	Get_Report	Return it
Class	Get_Idle	Return it
Class	Get_Protocol	Do nothing, we are not a boot device
Class	Set_Report	Do it
Class	Set_Idle	Do it
Class	Set_Protocol	Do nothing, we are not a boot device
Class	all others	Report error via a Stall

* "Do Nothing" requires the correct handshaking protocol. If an "Unknown" request is received, we must stall our request queue.

Example 1—Step 3: Implement Microcontroller Code

Now we're ready to implement the microcontroller code. Any of the microcontrollers presented so far could be used here. Each has a different SIE interface that requires slightly different software, but the overall structure of the solution will be the same.

For this example, I'll use the Anchor Chips EZ-USB because it is the easiest to explain. This microcontroller has an integrated USB port. (In Example 2, I'll substitute the Philips I2C-based peripheral but otherwise use the same simple design.)

I have divided the code into six modules (Figure 6-4):

- Three modules define program variables and fixed constants: DECLARE, DTABLE, and VECTOR.
- Three modules contain code: MAIN, INTERRUPT, and TIMER.

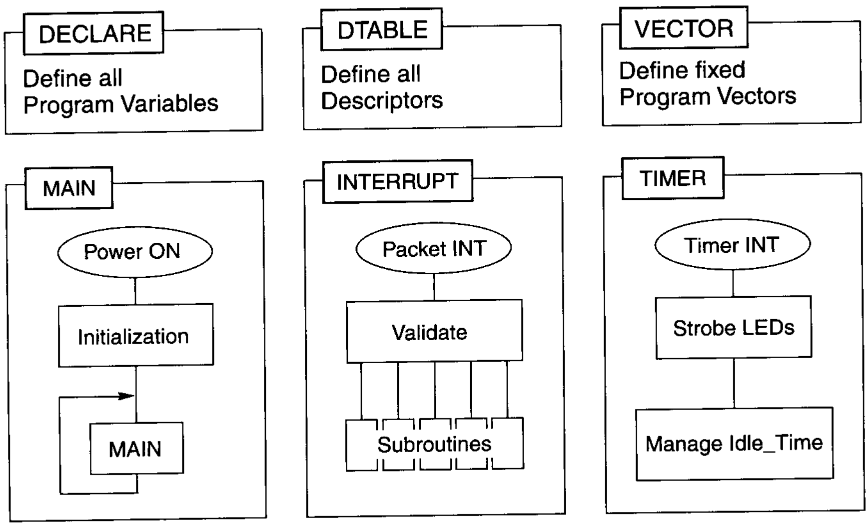


Figure 6-4. Overview of the I/O device firmware

The MAIN module receives control following power-on. After initializing the program variables and microcontroller peripherals, the module executes the MAIN task forever. The MAIN module receives buffers from the PC host, interprets their contents, and acts upon them. This module also generates buffers for the PC host and supplies them on request.

NOTE

All of the examples in this chapter will be able to use the same INTERRUPT module, because buffer interpretation and creation are handled in MAIN.

The INTERRUPT module does most of the work for the first example, because this module deals with the interrupts signaled by the SIE. It handles all standard requests and most of the HID class requests. Two of the HID class requests, SetReport and GetReport, result in data buffers that are passed to the MAIN module for processing.

Figure 6-5 shows how the EZ-USB responds to a control read transfer. The EZ-USB has a particularly smart SIE that validates received packets, generates the correct handshake packet, and interrupts the protocol microcontroller only after a good transaction has been received. The SIE generates separate interrupts for each transaction type and vectors the microcontroller, via a jump table, to the correct interrupt service routine. The EZ-USB also has multiple buffers to simplify the handling of control transfers.

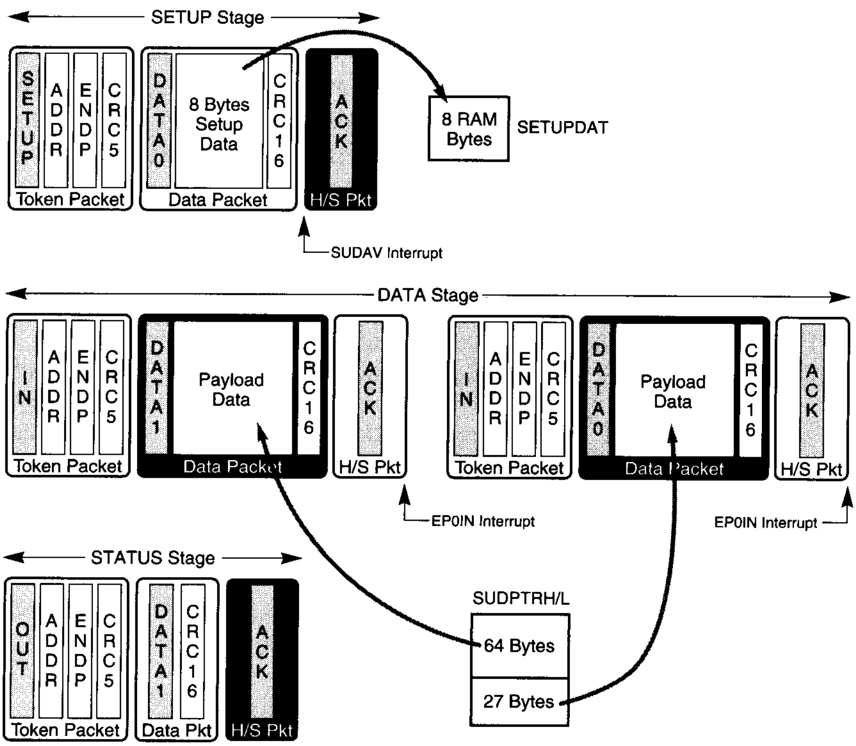


Figure 6-5. Operation of a control read transfer

The TIMER module is interrupt-driven. It has two tasks: (1) manage the LED ON time and (2) manage Idle_Time. In this first example the LEDs are strobed on one at a time, thus allowing the total power consumption of the example to stay below 100 mA. The TIMER module also handles Idle_Time processing, which allows and disallows the I/O device to respond to a GetReport request as specified for HID devices.

The protocol microcontroller will be interrupted by SIE once a valid control request has been received in the setup buffer. It is the responsibility of the protocol controller to service this request. The request types, detailed in Chapter 2, can be divided into three categories:

- Send data to the PC host.
- Receive data from the PC host.
- Handshake with the PC host.

Figure 6-5 shows the first case of sending data to the PC host. The protocol controller selects the data to send and places its address in a 16-bit SUDPTR register. Writing the low byte of this register prompts the SIE to send the data to the host. If the data is longer than 64 bytes (the maximum size of EP0 on the EZ-USB), then the SIE will send multiple data packets using DATA1 and DATA0 headers alternately. The SIE will also respond with an ACK to the PC host handshake. Because of the richness of the SIE implementation, there is not a lot for the protocol controller to do.

Receiving data from the PC host via an OUT transfer is similarly straightforward (Figure 6-6). The SIE notes the length of the data to be sent by reading bytes 6 and 7 of the SETUP DATA buffer. The SIE then copies received bytes into the OUT0BUF until the required length is reached, generates an EP0OUT interrupt to the protocol controller, and manages a handshake sequence to the PC host.

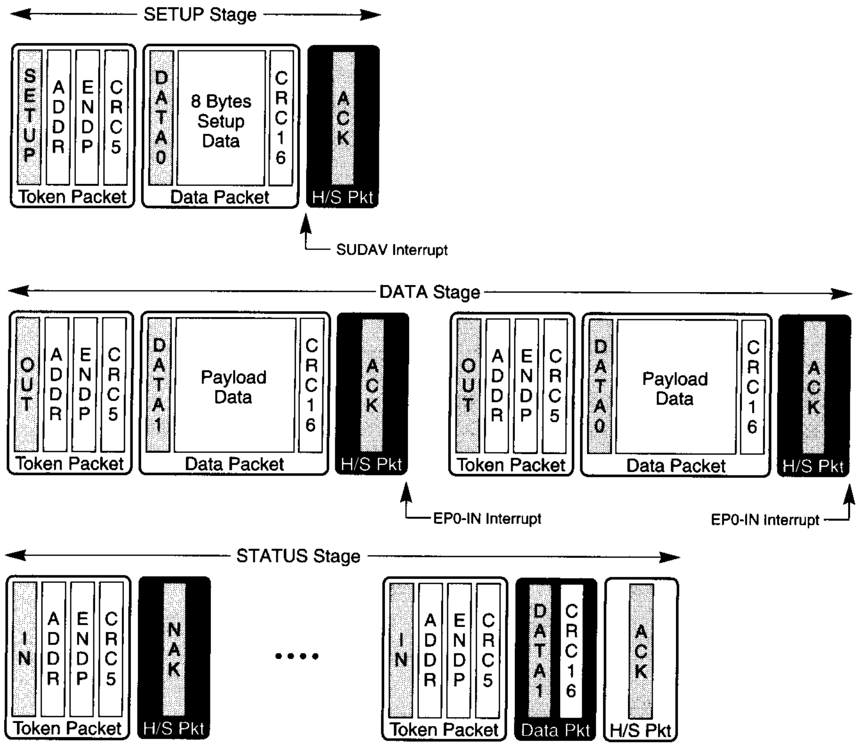
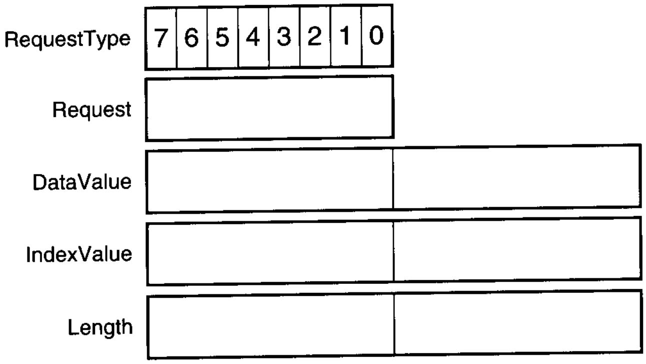


Figure 6-6. Responding to a control write

Thus, to implement an enumeration phase with the SIE interface in this example, the code needs to interpret SETUPDAT after a SUDAV (SetUp_Data_Available) interrupt and respond with data packets or absorb and react to data packets. This example accepts Standard and Class Requests for three recipients: Device, Interface, and Endpoint. The bit field encoding of **RequestType** lets us easily identify requests that are invalid for us. Figure 6-7 (repeated here from Chapter 2 for your convenience) shows that a 1 in bit positions 6, 4, 3, or 2 is not allowed, and our action would be to stall Endpoint 0. Having eliminated the majority of the invalid requests, this example creates an index (Figure 6-8) into the CommandTable using the valid bits from **RequestType** combined with the defined bits of **Request**. Figure 6-9 shows the CommandTable itself.



Definition of Request Type Bit Fields

Bit 7	0 = Transfer Host Data to Device		1 = Transfer Device Data to Host	
Bit 6-5	00 = Standard	01 = Class	10 = Vendor	11 = Reserved
Bit 4-0	0000 = Device All other codes are Reserved	00001 = Interface	00010 = Endpoint	00011 = Other

Figure 6-7. Format of a PC host request

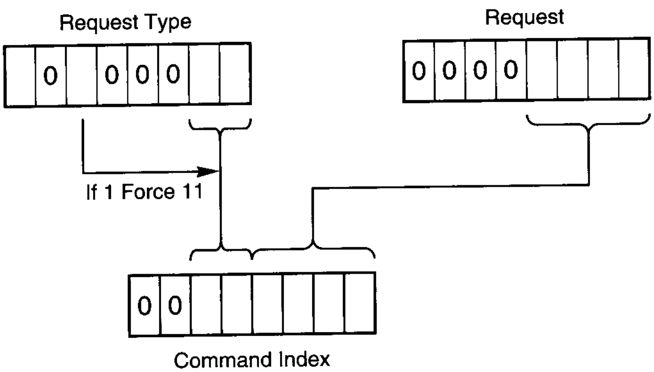


Figure 6-8. Creating an index into the CommandTable

NOTE

The CommandTable in Figure 6-9 includes all possible requests that an HID I/O device needs to respond to, so this approach will apply to all other examples in this chapter.

Requests that are not valid for this particular example will be routed to the INVALID routine. Note that 11 in bits 1:0 of **RequestType** is not valid, so, for coding convenience, the top quarter of the CommandTable is used for HID commands.

```
CommandTable:                          ; Table Index
;First 16 commands are for the Device
        Device_Get_Status              ; 00
        Device_Clear_Feature           ; 01
        Invalid                        ; 02
        Device_Set_Feature             ; 03
        Invalid                        ; 04
        Device_Set_Address             ; 05
        Device_Get_Descriptor          ; 06
        Device_Set_Descriptor          ; 07
        Device_Get_Configuration       ; 08
        Device_Set_Configuration       ; 09
        Invalid                        ; 0A
        Invalid                        ; 0B
        Invalid                        ; 0C
        Invalid                        ; 0D
        Invalid                        ; 0E
        Invalid                        ; 0F
;Next 16 commands are for the Interface
        Interface_Get_Status           ; 10
        Interface_Clear_Feature        ; 11
        Invalid                        ; 12
        Interface_Set_Feature          ; 13
        Invalid                        ; 14
        Invalid                        ; 15
        Class_Get_Descriptor           ; 16
        Class_Set_Descriptor           ; 17
        Invalid                        ; 18
        Invalid                        ; 19
        Interface_Get_Interface        ; 1A
        Interface_Set_Interface        ; 1B
        Invalid                        ; 1C
        Invalid                        ; 1D
        Invalid                        ; 1E
        Invalid                        ; 1F
;Next 16 commands are for the Endpoint
        Endpoint_Get_Status            ; 20
        Endpoint_Clear_Feature         ; 21
        Invalid                        ; 22
        Endpoint_Set_Feature           ; 23
        Invalid                        ; 24
        Invalid                        ; 25
        Invalid                        ; 26
        Invalid                        ; 27
        Invalid                        ; 28
        Invalid                        ; 29
        Invalid                        ; 2A
        Invalid                        ; 2B
```

```
            Endpoint_Sync_Frame         ; 2C
            Invalid                     ; 2D
            Invalid                     ; 2E
            Invalid                     ; 2F
;Next 16 commands are Class Requests
            Invalid                     ; 30
            Get_Report                  ; 31
            Get_Idle                    ; 32
            Get_Protocol                ; 33
            Invalid                     ; 34
            Invalid                     ; 35
            Invalid                     ; 36
            Invalid                     ; 37
            Invalid                     ; 38
            Set_Report                  ; 39
            Set_Idle                    ; 3A
            Set_Protocol                ; 3B
            Invalid                     ; 3C
            Invalid                     ; 3D
            Invalid                     ; 3E
            Invalid                     ; 3F
```

Figure 6-9. CommandTable for requests to HID I/O device

INTERRUPT Module Code

We must write a subroutine for each command. Many of these routines are similar but operate on different data. Rather than describe each routine here, I've added extra comments to the code for the INTERRUPT module (Figure 6-10).

```
; This module services requests from the SIE
;
        CSEG
ServiceSetupPacket:
        MOV     DPTR, #SETUPDAT         ; Point to Setup Packet data
        MOVX    A, @DPTR                ; Get the RequestType
        MOV     C, ACC.7                ; Bit 7 = 1 means IO device needs to send data to PC Host
        MOV     SendData, C
        ANL     A, #01011100b           ; IF RequestType[6.4.3.2] = 1 THEN goto BadRequest
        JNZ     BadRequest
        MOVX    A, @DPTR                ; IF RequestType[1&0] = 1 THEN goto BadRequest
        MOV     C, ACC.0
        ANL     C, ACC.1
        JC      BadRequest
        JNB     ACC.5, NotB5            ; IF RequestType[5] = 1 THEN RequestType[1,0] = [1,1]
        MOV     A, #00000011b
NotB5:  ANL     A, #00000011b           ; Set CommandIndex[5,4] = RequestType[1,0]
        SWAP    A
        MOV     Temp, A                 ; Save HI nibble of CommandIndex
                                        ; Set CommandIndex[3,0] = Request[3,0]
        INC     DPTR                    ; Point to Request
        MOVX    A, @DPTR
        ANL     A, #00001111b           ; Only 13 are defined today, handle in table
        ORL     A, Temp
        CALL    CorrectSubroutine       ; goto CommandTable(CommandIndex)
                                        ; Returns STALL=1 if a stall is required
        JB      STALL, BadRequest
        JNB     SendData, HandShake
        JB      IsDescriptor, LoadSUDPTR ; EZ-USB has a short cut for descriptors

        MOV     DPTR, #EP0InBuffer+2    ; Send data in ReplyBuffer
        MOV     R0, #ReplyBuffer+3
        MOV     Temp, #3                ; Copy maximum byte count
CopyRB: MOV     A, @R0
        MOVX    @DPTR, A
        DEC     DPL
        DEC     R0
        DJNZ    Temp, CopyRB
        MOV     A, @R0                  ; Get real byte count
SendEP0InBuffer:
        MOV     DPTR, #In0ByteCount
StartXfer: MOVX @DPTR, A                ; This write initiates the transfer
HandShake:                              ; Handshake with host
        MOV     Temp, #00000010b        ; Set HSNAK to tell the SIE that we're done
SetEP0Control:
        MOV     DPTR, #EP0Control
        MOVX    A, @DPTR
        ORL     A, Temp
        MOVX    @DPTR, A
        RET
```

```
LoadSUDPTR:                              ; Send the data pointed to by DPTR
        MOV     Temp, DPL
        MOV     A, DPH
        MOV     DPTR, #SUDPTR
        MOVX    @DPTR, A
        MOV     A, Temp
        INC     DPTR
        JMP     StartXfer
BadRequest:                              ; Invalid Request was received
        MOV     Temp, #00000011b         ; Set EP0STALL and HSNAK
        JMP     SetEP0Control

NextDPTR:                                ; Returns (DPTR + byte DPTR is pointing to)
        MOVX    A, @DPTR
BumpDPTR:                                ; Returns (DPTR + ACC)
        ADD     A, DPL
        MOV     DPL, A
        JNC     Skip
        INC     DPH                      ; Need 16 bit arithmetic here
Skip:   RET

CorrectSubroutine:                       ; Jump to the subroutine that DPTR is pointing to
        MOV     DPTR, #CommandTable
        CALL    BumpDPTR                 ; Point to entry
        MOVX    A, @DPTR                 ; Get the offset
        MOV     DPTR, #CommandTable
        CALL    BumpDPTR                 ; Get the routine address
        PUSH    DPL                      ; Create a RETURN address on stack
        PUSH    DPH                      ; Note: JMP @A+DPTR not used since ...
        MOV     R0, #ReplyBuffer+2       ; ... A and DPTR needed in subroutines
        CLR     A
        MOV     @R0, A                   ; Clear ReplyBuffer
        DEC     R0
        MOV     @R0, A
        DEC     R0
        MOV     @R0, #1                  ; Default non-descriptor reply
        MOV     DPTR, #SETUPDAT+2        ; Point to LOW(wValue)
        MOVX    A, @DPTR                 ; Many of the routines need these
        MOV     B, A                     ; LOW(wValue) in B
        INC     DPTR
        MOVX    A, @DPTR                 ; HIGH(wValue) in A
        CLR     STALL
        CLR     IsDescriptor
        RET                              ; Go to service routine

; Since the table only contains byte offsets, it is important that all these routines are
; within one page (100H) of CommandTable
;
CommandTable:
; First 16 commands are for the Device
        DB Device_Get_Status - CommandTable
        DB Device_Clear_Feature - CommandTable
        DB Invalid - CommandTable
        DB Device_Set_Feature - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable        ; SIE implements Device_Set_Address
        DB Get_Descriptor - CommandTable
        DB Set_Descriptor - CommandTable
        DB Get_Configuration - CommandTable
```

```
        DB Set_Configuration - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
; Next 16 commands are for the Interface
        DB Interface_Get_Status - CommandTable
        DB Interface_Clear_Feature - CommandTable
        DB Invalid - CommandTable
        DB Interface_Set_Feature - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Get_Class_Descriptor - CommandTable
        DB Set_Class_Descriptor - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Get_Interface - CommandTable
        DB Set_Interface - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
; Next 16 commands are for the Endpoint
        DB Endpoint_Get_Status - CommandTable
        DB Endpoint_Clear_Feature - CommandTable
        DB Invalid - CommandTable
        DB Endpoint_Set_Feature - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Endpoint_Sync_Frame - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
; Next 16 commands are Class Requests
        DB Invalid - CommandTable
        DB Get_Report - CommandTable
        DB Get_Idle - CommandTable
        DB Get_Protocol - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Set_Report - CommandTable
        DB Set_Idle - CommandTable
        DB Set_Protocol - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
        DB Invalid - CommandTable
;
```

```
; Many requests are INVALID for this example
Get_Protocol:                                   ; We are not a Boot device
Set_Protocol:                                   ; We are not a Boot device
Set_Descriptor:                                 ; Our Descriptors are static
Set_Class_Descriptor:                           ; Our Descriptors are static
Set_Interface:                                  ; We only have one Interface
Get_Interface:                                  ; We do not have an Alternate setting
Device_Set_Feature:                             ; We have no features that can be set or cleared
Interface_Set_Feature:                          ; We have no features that can be set or cleared
Endpoint_Set_Feature:                           ; We have no features that can be set or cleared
Device_Clear_Feature:                           ; We have no features that can be set or cleared
Interface_Clear_Feature:                        ; We have no features that can be set or cleared
Endpoint_Clear_Feature:                         ; We have no features that can be set or cleared
Endpoint_Sync_Frame:                            ; We are not an Isochronous device

Invalid:                                        ; Invalid Request made, STALL the Endpoint
        SETB    STALL
Reply:  RET

Set_Report:                                     ; Host wants to sent us a Report.
; The ONLY case in this example where host sends data to us
        JNB     Configured, Invalid             ; Need to be Configured to do this command
        MOV     DPTR, #OUT07IEN                 ; Enable EP0OUT interrupt so that MAIN will ...
        MOVX    A, @DPTR                        ; ... receive the Report
        ORL     A, #00000001b
        MOVX    @DPTR, A
        RET
Get_Report:                                     ; Host wants to know our Buttons Value
        JNB     Configured, Invalid             ; Need to be Configured to do this command
        MOV     A, Old_Buttons
        INC     R0                              ; Point to ReplyBuffer(1)
        MOV     @R0, A
        RET
Set_Idle:                                       ; Host wants to tell us how often we should talk
        JNB     Configured, Invalid             ; Need to be Configured to do this command
        MOV     Idle_Time, A
        RET                                     ; Handshake with host
Get_Idle:                                       ; Host must have forgotten what he told us to do
        JNB     Configured, Invalid             ; Need to be Configured to do this command
        INC     R0                              ; Point to ReplyBuffer(1)
        MOV     @R0, Idle_Time
        RET
Get_Configuration:
        JNB     Configured, Reply
                                                ; If configured return a 1 (via Device_Get_Status)
Device_Get_Status:                              ; Only two bits of Device Status are defined
        INC     R0                              ; Point to ReplyBuffer(1)
        MOV     @R0, #1                         ; Bit 1=Remote Wakeup(=0), Bit 0=Self Powered(=1)
        RET
Interface_Get_Status:                           ; Interface Status is currently defined as 0
Endpoint_Get_Status:
        MOV     @R0, #2
        RET
```

```
Set_Configuration:                              ; Valid values are 0 and 1
        MOV     A, B                            ; Get LOW(wValue)
        JZ      Deconfigured
        DEC     A
        JNZ     Invalid
        SETB    Configured
        RET
Deconfigured:
        CLR     Configured
        RET
Get_Descriptor:                                 ; Host wants to know who/what we are
        SETB    IsDescriptor
        DEC     A                               ; Valid Values are 1, 2 and 3
        MOV     DPTR, #DeviceDescriptor
        JZ      Reply
        DEC     A
        MOV     DPTR, #ConfigurationDescriptor
        JZ      Reply
        DEC     A
        JNZ     Invalid
        MOV     A, B                            ; Get string index
        CLR     C
        SUBB    A, #5                           ; Only have 4 strings
        JNC     Invalid                         ; Asked for a string I don't have
        MOV     A, B
        MOV     DPTR, #String0                  ; Now point to String 0
NextString:
        JZ      FixUpthenReply
        MOV     B, A                            ; Save string index
        CALL    NextDPTR
        MOV     A, B
        DEC     A
        JMP     NextString                      ; Check if we are there yet
Get_Class_Descriptor:                           ; Valid values are 21H, 22H, 23H for Class Request
        SETB    IsDescriptor
        CLR     C
        SUBB    A, #21H
        MOV     DPTR, #HIDDescriptor
        JZ      Reply
        DEC     A
        MOV     DPTR, #ReportDescriptor
        JZ      Reply
;       DEC     A                               ; This example does not use Physical Descriptors
;       JZ      Send_Physical_Descriptor
        JMP     Invalid
;
; Error check: this MUST be on within a page of CommandTable
WithinSamePage EQU $ - CommandTable
;
FixUpthenReply:                                 ; EZ-USB Rev D has a String Descriptor bug
                                                ; Need to fill the IN0BUF myself
; Note, the version of Dscope that I am using does not support EZ-USB features
; such as Dual DPTR's or the PageReg at SFR 92H, so do the move the old way
        MOVX    A, @DPTR                        ; Get the string length
        MOV     R7, A                           ; Save counter
        MOV     B, A
        MOV     R0, #80H
```

```
Copy1:  MOVX    A, @DPTR                ; First copy the string to low memory
        MOV     @R0, A
        INC     DPTR
        INC     R0
        DJNZ    R7, Copy1
        MOV     A, B
        MOV     R7, A
        MOV     DPTR, #EP0InBuffer
        MOV     R0, #80H
Copy2:  MOV     A, @R0
        MOVX    @DPTR, A
        INC     DPTR
        INC     R0
        DJNZ    R7, Copy2               ; Copy the String Descriptor into IN0BUF
; Fixup complete, get back to the program flow
        POP     ACC                     ; Get rid of the return address
        POP     ACC
        MOV     A, B                    ; Retrieve byte count
        JMP     SendEP0InBuffer
```

Figure 6-10. INTERRUPT module

MAIN Module Code

Figure 6-11 shows the MAIN module. Most of the module consists of system and variable initialization. This initialization code will be reusable in many of the other examples. The MAIN module also includes the service routines for processing the Input Report and creating the Output Report—these routines are the ones that will change with each of the examples.

```
; This module initializes the microcontroller then executes MAIN forever
;
Reset:  MOV     SP, #STACK-1            ; Initialize the Stack
        MOV     DPTR, #USBControl       ; Simulate a disconnect
        MOVX    A, @DPTR
        ANL     A, #11110011b           ; Clear DISCON, DISCOE
        MOVX    @DPTR, A
        CALL    Wait100msec             ; Give the host time to react
        MOVX    A, @DPTR                ; Reconnect with this new identity
        ORL     A, #00000110b           ; Set DISCOE to enable pullup resistor
        MOVX    @DPTR, A                ; Set RENUM so that 8051 handles USB requests
        CLR     A
        MOV     FLAGS, A                ; Start in Default state
TurnOffLEDs:
        MOV     LEDValue, A
        MOV     Old_Buttons, A
        MOV     LEDstrobe, #11H
Initialize4msecCounter:
        INC     A                       ; = 1
        MOV     Expired_Time, A
        MOV     Msec_counter, A
        CALL    InitializeIOSystem      ; Work around the Dscope monitor I/O needs
        CALL    InitializeInterruptSystem ; First initialize the USB level then the main level
; Initialization Complete.
        MAIN:   NOP                     ; Not much of a main loop for this example
        JMP     MAIN                    ; All actions are initiated by interrupts
; We are a slave, we wait to be told what to do

ProcessOutputReport:                    ; A Report has just been received
; The report is only one byte long in this first example
; It contains a new value for the LEDs
        MOV     DPTR, #EP0OutBuffer     ; Point to the Report
        MOVX    A, @DPTR                ; Get the Data
        MOV     LEDValue, A             ; Update the local variable
        RET
CreateInputReport:                      ; Called from TIMER which detected the need
; The report is only one byte long in this first example
; It contains a new value for the Buttons
        MOV     DPTR, #EP1InBuffer      ; Point to the buffer
        MOV     A, Old_Buttons          ; Get the Data
        MOVX    @DPTR, A                ; Update the Report
        MOV     DPTR, #IN1ByteCount
        MOV     A, #1
        MOVX    @DPTR, A                ; Endpoint 1 now 'armed', next IN will get data
        RET
```

Figure 6-11. MAIN module

TIMER Module Code

The TIMER module (Figure 6-12) receives constant regular interrupts and implements time-based tasks such as strobing the LEDs and counting down the Report-based Idle_Time. I used the Start-Of-Frame Interrupt because this was convenient.

```
; This module services the real time interrupt
; It is also responsible for the "real world" buttons and lights
;
; Get a Real Time interrupt every One millisecond (using SOF interrupt)
; HID devices work on a 4 millisecond timer
; We have two tasks
;           a) Strobe the LEDs (only one is ever really on, saves power)
;           b) Check Report Idle_Time to see if we have a report to send
ServiceTimerRoutine:
            DJNZ    Msec_counter, Done          ; Only need to check every 4msec
            MOV     Msec_counter, #4            ; Reinitialize
;
; LED task
; Light the next LED in sequence
            MOV     A, LEDStrobe                ; Get the current enabling pattern
            RL      A
            MOV     LEDStrobe, A                ; Save for next time
            ANL     A, LEDValue                 ; Get the LED image
            CALL    WriteLEDs                   ; Update the real world
;
;Handle Idle_Time task
; Check if Buttons have changed or Idle_Time has expired
            CALL    ReadButtons                 ; Have the buttons changed in the last 4msec?
            MOV     LEDValue, A
            MOV     Temp, A
            XRL     A, Old_Buttons
            JZ      NoChange
            MOV     Old_Buttons, Temp           ; Update current button position
            JMP     ResetExpiredTime
; Buttons have not changed, Check Idle_Time
NoChange:
            MOV     A, Idle_Time
            JZ      Done                        ; Check special case = 0 (only on change)
; IF Idle_Time <> 0, THEN check the expired time
            DJNZ    Expired_Time, Done
; Idle_Time has expired, reinitialize and enable response to IN 1
ResetExpiredTime:
            MOV     Expired_Time, Idle_Time
            CALL    CreateInputReport
Done:       RET

; Talk to the "real world" buttons and lights
ReadButtons:
            MOV     DPTR, #PortA_Pins
            MOVX    A, @DPTR
            RET
WriteLEDs:
            CPL     A                           ; 0 = LED on
            MOV     DPTR, #PortB_Out
            MOVX    @DPTR, A
            RET
```

Figure 6-12. TIMER module

DECLARE Module Code

The DECLARE module is straightforward to code (Figure 6-13).

```
; This module declares the variables and constants used in the program
;
; Declare Special Function Registers used
EI              DATA        0A8H
EIE             DATA        0E8H            ; EZ-USB specific
EXIF            DATA        091H            ; EZ-USB specific
EICON           DATA        0D8H            ; EZ-USB specific
;
; "External" memory locations used, EZ-USB specific
SETUPDAT        EQU         07FE8H
SUDPTR          EQU         07FD4H
EP0Control      EQU         07FB4H
EP0InBuffer     EQU         07F00H
EP0OutBuffer    EQU         07EC0H
EP1InBuffer     EQU         07E80H
IN0ByteCount    EQU         07FB5H
IN1ByteCount    EQU         07FB7H
IN07IEN         EQU         07FACH
IN07IRQ         EQU         07FA9H
OUT07IEN        EQU         07FADH
OUT07IRQ        EQU         07FAAH
USBIEN          EQU         07FAEH
USBIRQ          EQU         07FABH
USBControl      EQU         07FD6H
PortA_PINS      EQU         07F99H
PortB_PINS      EQU         07F9AH
PortA_Config    EQU         07F93H
PortB_Config    EQU         07F94H
PortB_OUT       EQU         07F97H
PortA_OE        EQU         07F9CH
PortB_OE        EQU         07F9DH
;
; Byte Variables
                DSEG AT 20H
FLAGS:          DS          1               ; This register is bit-addressable
MonitorSpace:   DS          1FH             ; Used by Dscope
Temp:           DS          1               ; A temporary working register
Msec_counter:   DS          1               ; Counts milliseconds
Old_Buttons:    DS          1               ; Stores current button positions
Idle_Time:      DS          1               ; The time the PC host wants us to wait
Expired_Time:   DS          1               ; A downcounter for timed Reports
LEDstrobe:      DS          1               ; Strobe to turn on one LED at a time
LEDValue:       DS          1               ; Stores current LED values
ReplyBuffer:    DS          3               ; First byte is Count
STACK:          DS          20
; Bit Variables
Configured      EQU         FLAGS.0         ; Is this device configured
STALL           EQU         FLAGS.1         ; Need to STALL endpoint 0
SendData        EQU         FLAGS.2         ; Need to send data to PC Host
IsDescriptor    EQU         FLAGS.3         ; Enable a shortcut reply
;
```

Figure 6-13. DECLARE module

DTABLE Module Code

The DTABLE module is straightforward to code (Figure 6-14).

```
; This module declares the descriptors
;
; This example has one Device Descriptor with:
;         One Configuration - single IN port and single OUT port
;         One Interface - there is only one method of accessing the ports
;         One HID Descriptor - to make PC host software simpler
;         One Endpoint Descriptor - for HID Input Reports
;         One Report Descriptor - one byte IN and one byte OUT reports
;         Multiple Sting Descriptors - to aid the user
;
        CSEG
DeviceDescriptor:
        DB      18, 1                   ; Length, Type
        DW      101H                    ; USB Rev 1.1
        DB      0, 0, 0                 ; Class, Subclass and Protocol
        DB      64                      ; EP0 size
        DW      4242H, 1, 1             ; Vendor ID, Product ID and Version
        DB      1, 2, 0                 ; Manufacturer, Product & Serial# Names
        DB      1                       ; #Configs
ConfigurationDescriptor:
        DB      9, 2                    ; Length, Type
        DB      LOW(ConfigLength), HIGH(ConfigLength)
        DB      1, 1, 3                 ; #Interfaces, Configuration#, Config. Name
        DB      10000000b               ; Attributes = Bus Powered
        DB      50                      ; Max. Power is 50x2 = 100mA
InterfaceDescriptor:
        DB      9, 4                    ; Length, Type
        DB      0, 0, 1                 ; No alternate setting, HID uses EP1
        DB      3                       ; Class = Human Interface Device
        DB      0, 0                    ; Subclass and Protocol
        DB      4                       ; Interface Name
HIDDescriptor:
        DB      9, 21H                  ; Length, Type
        DB      0, 1                    ; HID Class Specification compliance
        DB      0                       ; Country localization (=none)
        DB      1                       ; Number of descriptors to follow
        DB      22H                     ; And it's a Report descriptor
        DB      LOW(ReportLength), HIGH(ReportLength)
EndpointDescriptor:
        DB      7, 5                    ; Length, Type
        DB      10000001b               ; Address = IN 1
        DB      00000011b               ; Interrupt
        DB      64, 0                   ; Maximum packet size (this example only uses 1)
        DB      100                     ; Poll every 0.1 seconds
ConfigLength    EQU $ - ConfigurationDescriptor

ReportDescriptor:                       ; Generated with HID Tool, copied to here
        DB      6, 0, 0FFH              ; Usage_Page (Vendor Defined)
        DB      9, 1                    ; Usage (I/O Device)
        DB      0A1H, 1                 ; Collection (Application)
        DB      19H, 1                  ;   Usage_Minimum (Button 1)
        DB      29H, 8                  ;   Usage_Maximum (Button 8)
        DB      15H, 0                  ;   Logical_Minimum (0)
        DB      25H, 1                  ;   Logical_Maximum (1)
```

```
                DB      75H, 1                  ;  Report_Size (1)
                DB      95H, 8                  ;  Report_Count (8)
                DB      81H, 2                  ;  Input (Data,Var,Abs)
                DB      19H, 1                  ;  Usage_Minimum (Led 1)
                DB      29H, 8                  ;  Usage_Maximum (Led 8)
                DB      91H, 2                  ;  Output (Data,Var,Abs)
                DB      0C0H                    ; End_Collection
ReportLength            EQU $-ReportDescriptor

String0:                                        ; Declare the UNICODE strings
                DB      4, 3, 9, 4              ; Only English language strings supported
String1:                                        ; Manufacturer
                DB      (String2-String1),3     ; Length, Type
                DB      "U",0,"S",0,"B",0," ",0,"D",0,"e",0,"s",0,"i",0,"g",0,"n",0," ", 0
                DB      "B",0,"y",0," ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
String2:                                        ; Product Name
                DB      (String3-String2),3
                DB      "B",0,"u",0,"t",0,"t",0,"o",0,"n",0,"s",0," ",0
                DB      "&",0," ",0,"L",0,"i",0,"g",0,"h",0,"t",0,"s",0
String3:                                        ; Configuration Name
                DB      (String4-String3),3
                DB      "S",0,"i",0,"m",0,"p",0,"l",0,"e",0," ",0,"I",0,"/",0
                DB      "O",0," ",0,"D",0,"e",0,"v",0,"i",0,"c",0,"e",0
String4:                                        ; Interface Name
                DB      (EndOfDescriptors-String4),3
                DB      "F",0,"i",0,"r",0,"s",0,"t",0," ",0,"H",0,"I",0,"D",0
                DB      " ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
EndOfDescriptors:
                DW      0                       ; Backstop for String Descriptors
```

Figure 6-14. DTABLE module

VECTOR Module Code

The VECTOR module is straightforward to code (Figure 6-15).

```
; This module contains all of the interrupt vector declarations and
; the first level interrupt servicing (register save, call subroutine,
; clear interrupt source, restore registers, return)
; Suspend and Resume are handled totally in this module
;
; A Reset sends us to Program space location 0
        CSEG AT 0                       ; Code space
        USING 0                         ; Reset forces Register Bank 0
        LJMP    Reset
;
; The interrupt vector table is also located here
; EZ-USB has two levels of USB interrupts:
; 1-the main level is described in this table (at ORG 43H)
; 2-there are 21 sources of USB interrupts and these are described in USB_ISR
; This means that two levels of acknowledgement and clearing will be required
;       LJMP    INT0_ISR
;       ORG     0BH
;       LJMP    Timer0_ISR
;       ORG     13H
;       LJMP    INT1_ISR
;       ORG     1BH
;       LJMP    Timer1_ISR
;       ORG     23H
;       LJMP    UART0_ISR
;       ORG     2BH
;       LJMP    Timer2_ISR
;       ORG     33H
;       LJMP    WakeUp_ISR
;       ORG     3BH
;       LJMP    UART1_ISR
        ORG     43H
        LJMP    USB_ISR                 ; Auto Vector will replace byte 45H
;       ORG     4BH
;       LJMP    I2C_ISR
;       ORG     53H
;       LJMP    INT4_ISR
;       ORG     5BH
;       LJMP    INT5_ISR
;       ORG     63H
;       LJMP    INT6_ISR

        ORG     0E0H                    ; Keep out of the way of Tscope monitor

; When a feature is used insert the required interrupt processing here
; This example only used Endpoints 0 and 1 and also SOF for timing
Reserved:
INT0_ISR:
Timer0_ISR:
INT1_ISR:
Timer1_ISR:
UART0_ISR:
Timer2_ISR:
UART1_ISR:
I2C_ISR:
```

```
INT4_ISR:
INT5_ISR:
INT6_ISR:
SUTOK_ISR:
EP0In_ISR:
EP1In_ISR:
EP1Out_ISR:
EP2In_ISR:
EP2Out_ISR:
EP3In_ISR:
EP3Out_ISR:
EP4In_ISR:
EP4Out_ISR:
EP5In_ISR:
EP5Out_ISR:
EP6In_ISR:
EP6Out_ISR:
EP7In_ISR :
EP7Out_ISR:
Not_Used:                                   ; Should not get any of these
            RETI

ClearINT2:                                  ; Tell the hardware that we're done
            MOV     A, EXIF
            CLR     ACC.4                   ; Clear the Interrupt 2 bit
            MOV     EXIF, A
            RET

USBReset_ISR:                               ; Bus has been Reset, move to DEFAULT state
            PUSH    ACC
            CLR     Configured
CALL        ClearINT2
                                            ; No need to clear source of interrupt
            POP     ACC
            RETI

Suspend_ISR:                                ; SIE detected an Idle bus
            PUSH    PSW
            PUSH    ACC
            CLR     A
            CALL    WriteLEDs               ; Turn off the lights and ...
            MOV     A, PCON
            ORL     A, #1
            MOV     PCON, A                 ;          ... go to sleep!
            NOP
            NOP                             ; Wake up here due to a USBResume
            NOP
            CALL    ClearINT2
            POP     ACC
            POP     PSW
            RETI

WakeUp_ISR:                                 ; Not using external WAKEUP in this example
                                            ; So this must be due to a USBResume
            CLR     EICON.4                 ; Clear the wakeup interrupt source
            RETI
```

```
SUDAV_ISR:                              ; A Setup packet has been received
        PUSH    PSW                     ; Save Registers before the service routine
        PUSH    ACC
        PUSH    DPL
        PUSH    DPH
        CALL    ServiceSetupPacket
        CALL    ClearINT2
                                        ; Clear the source of the interrupt
        MOV     DPTR, #USBIRQ
        MOV     A, #00000001b
        MOVX    @DPTR, A
        POP     DPH                     ; Restore Registers
        POP     DPL
        POP     ACC
        POP     PSW
        RETI

SOF_ISR:                                ; A Start-Of-Frame packet has been received
        PUSH    PSW                     ; Save Registers before the service routine
        PUSH    ACC
        PUSH    DPL
        PUSH    DPH
        CALL    ServiceTimerRoutine
        CALL    ClearINT2
                                        ; Clear the source of the interrupt
        MOV     DPTR, #USBIRQ
        MOV     A, #00000010b
        MOVX    @DPTR, A
        POP     DPH                     ; Restore Registers
        POP     DPL
        POP     ACC
        POP     PSW
        RETI

EP0Out_ISR:                             ; An Set Report has been received
                                        ; The only data the PC Host sends in this example
        PUSH    PSW                     ; Save Registers before the service routine
        PUSH    ACC
        PUSH    DPL
        PUSH    DPH
        CALL    ProcessOutputReport
        MOV     DPTR, #OUT07IEN         ; Disable this interrupt now
        MOVX    A, @DPTR
        ANL     A, #11111110b
        MOVX    @DPTR, A
        CALL    ClearINT2
                                        ; Clear the source of the interrupt
        MOV     DPTR, #OUT07IRQ
        MOV     A, #00000001b
        MOVX    @DPTR, A
        POP     DPH                     ; Restore Registers
        POP     DPL
        POP     ACC
        POP     PSW
        RETI

        ORG     ($+0FFH) AND 0FF00H
                                        ; Auto Vector requires this on a page boundary
```

```
USB_ISR:LJMP    SUDAV_ISR
        DB      0                       ; Pad entries to 4 bytes
        LJMP    SOF_ISR
        DB      0
        LJMP    SUTOK_ISR
        DB      0
        LJMP    Suspend_ISR
        DB      0
        LJMP    USBReset_ISR
        DB      0
        LJMP    Reserved
        DB      0
        LJMP    EP0In_ISR
        DB      0
        LJMP    EP0Out_ISR
        DB      0
        LJMP    EP1In_ISR
        DB      0
        LJMP    EP1Out_ISR
        DB      0
        LJMP    EP2In_ISR
        DB      0
        LJMP    EP2Out_ISR
        DB      0
        LJMP    EP3In_ISR
        DB      0
        LJMP    EP3Out_ISR
        DB      0
        LJMP    EP4In_ISR
        DB      0
        LJMP    EP4Out_ISR
        DB      0
        LJMP    EP5In_ISR
        DB      0
        LJMP    EP5Out_ISR
        DB      0
        LJMP    EP6In_ISR
        DB      0
        LJMP    EP6Out_ISR
        DB      0
        LJMP    EP7In_ISR
        DB      0
        LJMP    EP7Out_ISR              ; End of Interrupt Vector tables
;
```

Figure 6-15. VECTOR module

Example 1—Step 4: Application Code

Figure 6-16 shows a screen shot of the Visual Basic application program. There are two sets of buttons—one is "soft" and the other is a copy of the real buttons. The "soft" LEDs are a copy of the real LEDs and respond the same way. The LEDs are operated by the two sets of buttons (A = soft buttons, B = real buttons) as:

A only
A OR B
A AND B
A XOR B
B only

This scheme allows software to control the LEDs, hardware (the real buttons) to control the LEDs, or a combination of both. The full source code is provided on the CD-ROM.

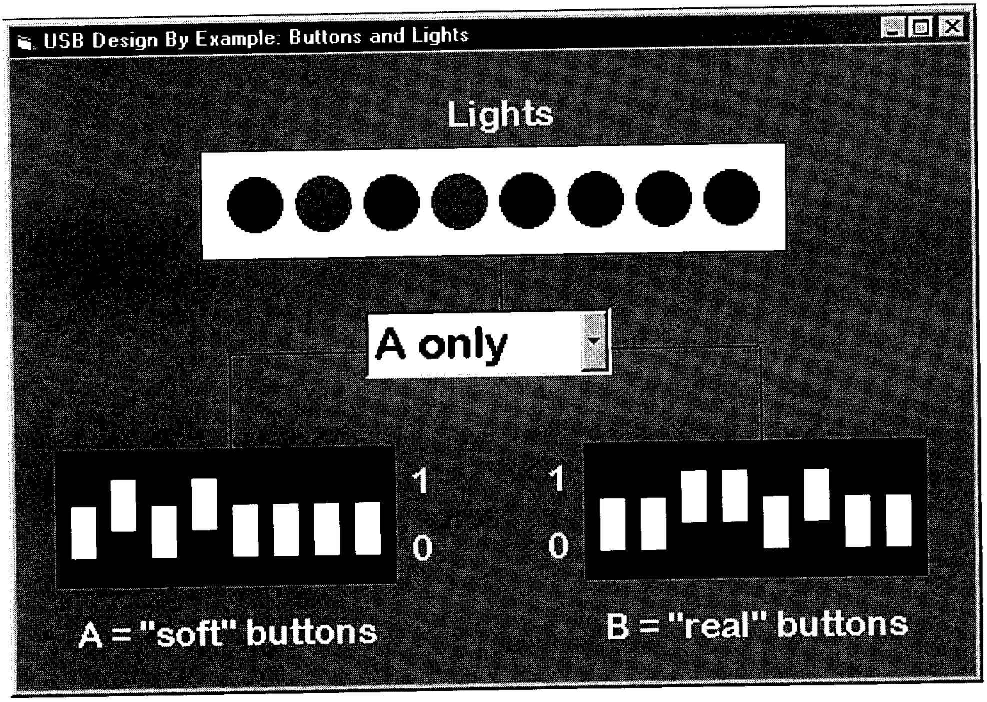


Figure 6-16. Buttons and lights application program

Example 1—Design Summary

The example was a simple 8-bit input port and an 8-bit output port so we could focus on the method. We have successfully used external hardware to change the operation of PC host software. We have also used software to change some hardware external to the PC. This is a significant milestone—we have PC host software interacting with a custom I/O board. Now that we have achieved this first step, we can investigate monitoring and controlling many other external hardware examples.

EXAMPLE 2: SIMPLE DESIGN, EXTERNAL USB PORT

Before extending our first, simple example, I want to reuse the example but substitute a different microcontroller so that a wider range of applications can be covered.

Example 2—Step 1: Design the Hardware

This example uses a generic microcontroller with an attached USB peripheral to implement the USB port (Figure 6-17). The microcontroller is an MCS51, and the USB peripheral in this example is the Philips PDIUSBD11, which has an I^{2}C interface. I have included a photograph of the Philips part in Figure 6-18 to show how small the circuitry for a USB connection can be.

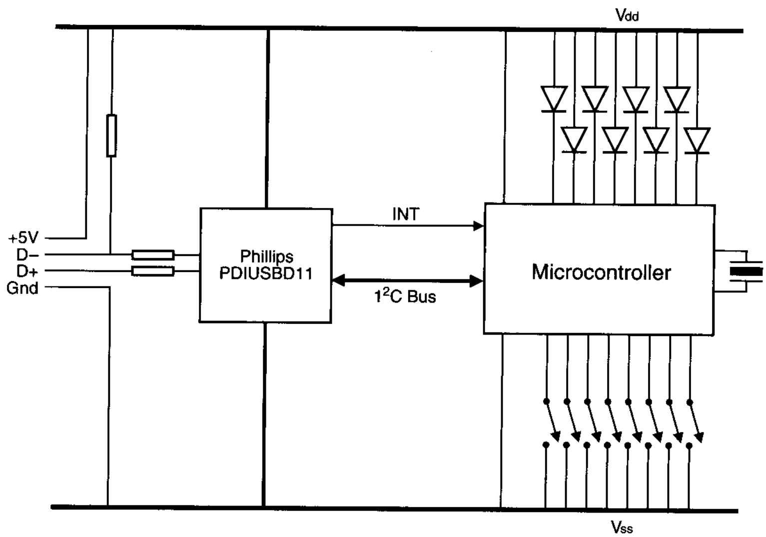


Figure 6-17. Microcontroller with external USB connection



Courtesy of Philips Semiconductors.

Figure 6-18. Philips PDIUSBD11 USB peripheral

Example 2—Step 2: Complete the Descriptors

This step is the same as in Example 1. No changes are required.

Example 2—Step 3: Implement Microcontroller Code

The SIE interface presented by the PDIUSBD11 consists of a single interrupt and a large set of registers that are accessed over an I^2C bus. The PDIUSBD11 is a slave I^2C device, and the microcontroller is the master. Many microcontrollers include an on-chip I^2C master device, but some do not—this interface is easy to "bit-bang" as described in Application Note 476 in the Chapter 6 directory on the CD-ROM. My example microcontroller has an on-chip I^2C master device. The PDIUSBD11 component requires some initialization that is added to our MAIN module, but most of the changes will be in the INTERRUPT module.

When a packet arrives, the PDIUSBD11 drives its interrupt line low. The microcontroller services this interrupt by first reading the interrupt status register of the PDIUSBD11 and then reading other registers to acquire the packet type and packet data information. In contrast with the EZ_USB implementation, which only generates an interrupt request on a successful transaction, the PDIUSBD11 (and many other devices) generates an interrupt on every packet. The sequences of packets were described in Chapter 2, and the service routine in this example must monitor and control the sequencing and check for all errors. This is a lot more work, and fortunately Philips provides example code for their component. I have modified this slightly so that it will generate the same

subroutine calls as our previous example, enabling the reuse of the request decoding and response code. The original Philips software and the extra software required for the Philips PDIUSBD11 are included on the CD-ROM. I have also included several other examples in the Chapter 6/Philips directory on the CD-ROM.

Example 2—Step 4: Application Code

This example uses the same application program as in Example 1. It is worthwhile to point out that the Visual Basic application program has **no knowledge** of the different hardware of Example 2. The application program talks to the **Interface** provided by the I/O device descriptors, and because both examples use the same descriptor tables, then the operation of the application software will be identical. This "hardware-independence" is a key USB attribute.

Example 2—Design Summary

A USB connection was added to an existing microcontroller simply by using one of the available USB peripheral components. I've used a serial connection (I^2C) because high performance is not required in the example. If higher performance is necessary, there are several parallel interface USB peripherals available (see Appendix A). The complete working code for an MCS51 microcontroller is demonstrated and included on the CD-ROM. This code can be ported to another microcontroller family if required (this is left as an exercise for the reader).

Example 2—Real-world Product Example

Figure 6-19 shows an interesting variation of this example. This diagram shows the Anchor Chips EZ-USB with its integrated USB **and** an I^2C -connected USB. Why have two USB connections? Well, we could connect each USB to a different PC host and thus create a PC-to-PC link! With some software added to this, we could create a potent high-speed link. Anchor Chips offers a product called EZ-Link (see Appendix A) that implements all of this for you.

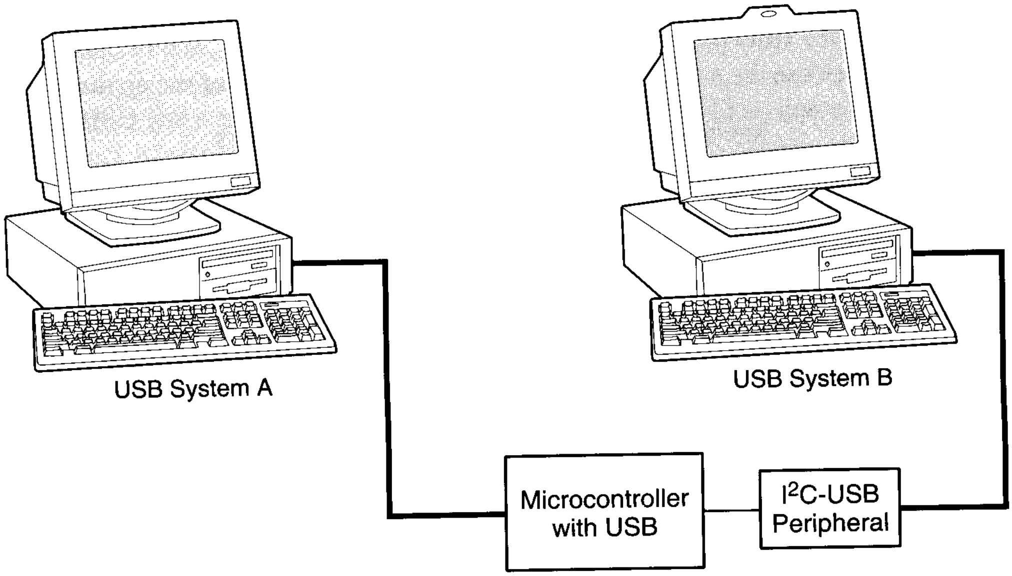


Figure 6-19. A microcontroller with two USB connections

EXAMPLE 3: ADDING MORE PORTS, MICROCONTROLLER-INDEPENDENT

Extending the number of I/O ports is very straightforward given the infrastructure we've built. The data buffers that the PC host and I/O device exchange will be longer, but this has almost no impact on the software we have written so far. The code is independent of the microcontroller and could thus apply to either Example 1 or 2.

The application program will generate more data that the I/O device needs to interpret.

Example 3—Step 1: Design the Hardware

I have chosen a simple extension of the buttons and lights example in the form of a "reader board." This example (Figure 6-20) expands from our single column of LEDs to make a 40-column x 7-row display. Each of the columns is enabled sequentially and strobes at a rate faster than 30 Hz, so that human persistence of vision will make them appear always on. Each column of seven LEDs consumes 150 mA if all LEDs are lit, so strobing the columns saves power. A transistor is required to sink each column's current, and a line driver is used to provide current for each row.

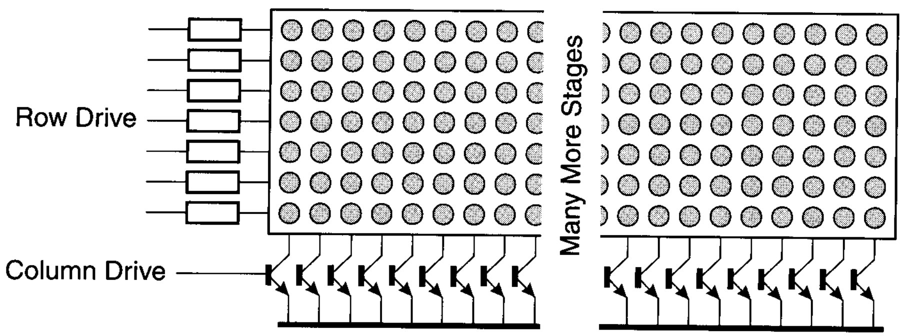


Figure 6-20. 40-column x 7-row reader board

To access the 40 individual columns, the microcontroller can use one of two basic methods: parallel decode or serial decode (Figure 6-21). Both methods have their benefits and drawbacks.

- The parallel decoder uses six buffered I/O lines and cascaded 3-to-8-line decoders to select a single line from an address on the six I/O lines. Unfortunately, all of the available components are built with the selected line active low, which means that inverters would need to be added before the transistor driver.
- The serial decoder uses a series of shift registers to rotate a "selection bit" across the columns. The shift registers have a CLR input that sets all outputs low so the drive transistors are turned off. A CLK input shifts the DataIN signal to the right. This DataIN signal would be high when the first column is selected and low at all other times. The disadvantage of the serial scheme, in this example, is that multiple columns could be accidentally selected because of a program bug—this would cause an excessive current draw.

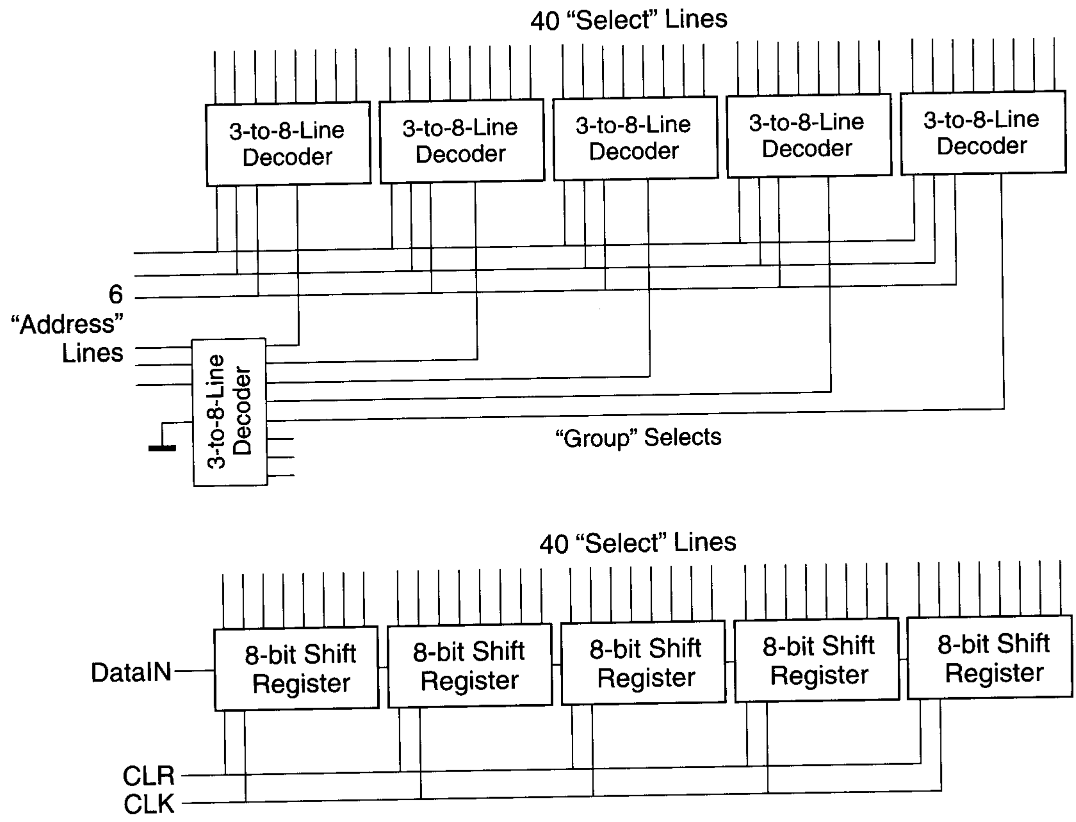


Figure 6-21. Parallel and serial selection schemes

The choice of parallel or serial enabling is a local one. Only the microcontroller needs to know which method is being used. The interface seen by the PC host application program is a 40-byte buffer. This hardware independence will be demonstrated again in Example 4.

Figure 6-22 shows the final circuit for Example 3; I chose a serial scheme to use with eight 5x7 dot-matrix LED components (see the data sheets in the Chapter 6 directory on the CD-ROM).

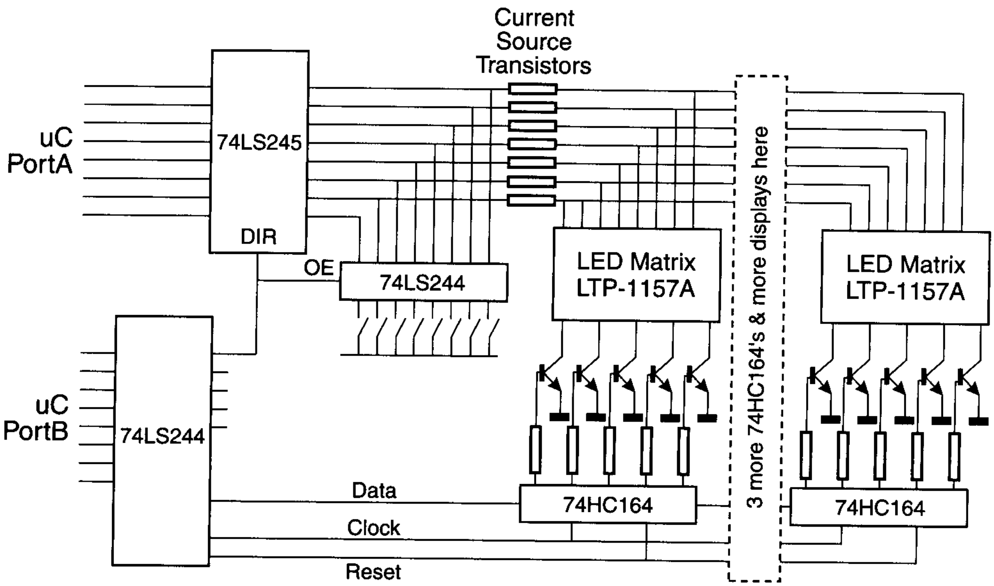


Figure 6-22. 40-column x 7-row reader board example

Example 3—Step 2: Complete the Descriptors

There are two minor changes to the descriptors:

- We change the product code and description in the device descriptor.
- The length of the Reports in the ReportDescriptor is increased.

Example 3—Step 3: Implement Microcontroller Code

Few changes are needed to convert our code from Example 1 to the reader board code:

- The ProcessOutputReport routine and the CreateInputReport routine must accommodate the extra report lengths.
- We extend the TIMER routine to handle the multiple columns.

That's it. The full source is available on the CD-ROM.

Example 3—Step 4: Application Code

The reader board example allows individual LEDs to be set ON or OFF. The program first reads the current display from the I/O device and allows editing that will be downloaded to the display later. The I/O device maintains the display after the application software has exited. Figure 6-23 shows the human interface design—it is extended from Example 1. The full source is available on the CD-ROM.

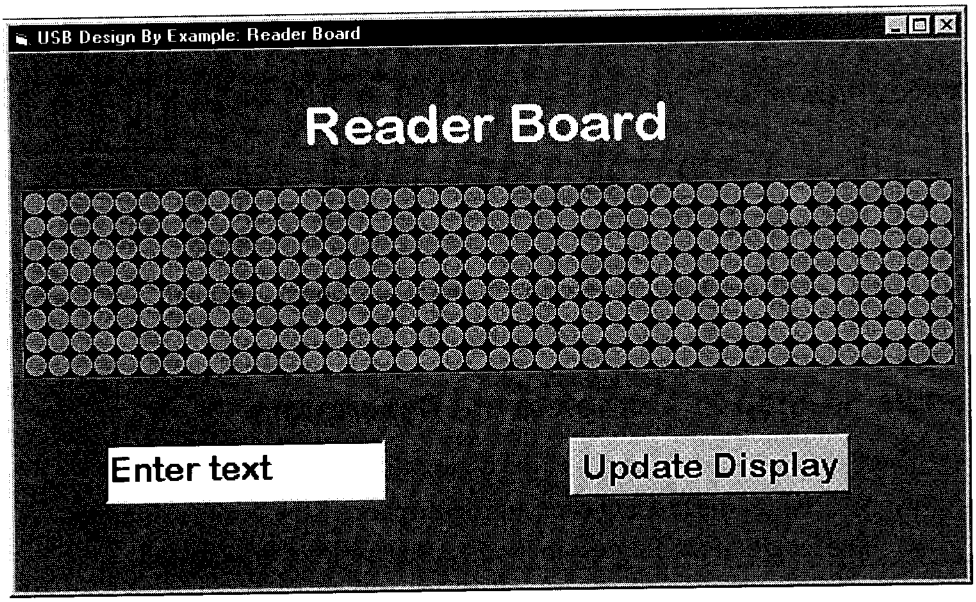


Figure 6-23. Example human interface design, reader board

Example 4: Adding Lots More Ports, Microcontroller-Independent

In this example we extend the number of I/O ports to be even more than in Example 3. The data buffers that the PC host and I/O device exchange will be longer, but this has almost no impact on the software we have written so far. To make this example more interesting, we shall add another configuration as described in step 2.

This example expands to a 120-column x 21-row reader board display of LEDs from the previous 40 columns (Figure 6-24). The design could be extended to support many more columns that could even "overflow" into multiple lines.

Example 4—Step 1: Design the Hardware

I recommend using a parallel drive solution—that is, run multiple, parallel blocks of the 40x7 display we have just designed. The row drivers need to latch their row data from the data bus so that each block can be driven with different row data. Although they are drawn as separate blocks in Figure 6-24 (to show the component interconnect), each horizontal block would be physically adjacent to the next to form a continuous display. One column of all blocks will be on at any one time, so higher-power column transistors will be required, and the total power dissipation will rise.

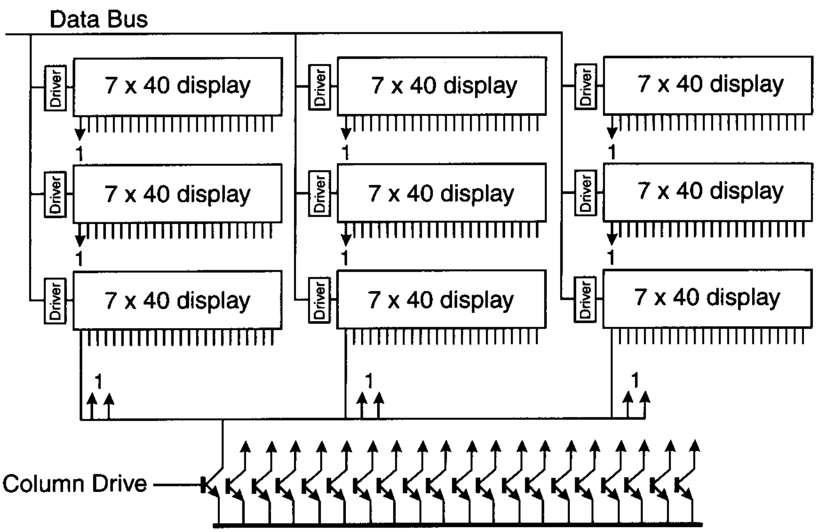


Figure 6-24. Large reader board example

Example 4—Step 2: Complete the Descriptors

It would be more convenient to be able to input ASCII text into the reader board. To do this, we need to translate the ASCII characters into a 5x7 dot pattern. While this could be done by the application program, it would be a better design choice to implement this hardware-dependent feature in the I/O device. Let's enhance our I/O device to support two configurations—a **raw data** mode and an **ASCII** mode.

The first step is to extend the descriptors to include two configuration descriptors (Figure 6-25). Each has an associated Interface Descriptor, but no added endpoints are specified because our communication with the PC host uses the HID report mechanism, which targets Endpoint 0.

```
; This module declares the descriptors
;
; This example has one Device Descriptor with:
;         Two Configurations – RAW and ASCII
;         Since the RAW and ASCII reports are different lengths need two HID descriptors
;         Multiple Sting Descriptors - to aid the user
;
        CSEG
DeviceDescriptor:
        DB      18, 1                   ; Length, Type
        DW      101H                    ; USB Rev 1.1
        DB      0, 0, 0                 ; Class, Subclass and Protocol
        DB      64                      ; EP0 size
        DW      4242H, 1, 1             ; Vendor ID, Product ID and Version
        DB      1, 2, 0                 ; Manufacturer, Product & Serial# Names
        DB      2                       ; #Configs
RAWConfigurationDescriptor:
        DB      9, 2                    ; Length, Type
        DB      LOW(RAWConfigLength), HIGH(RAWConfigLength)
        DB      1, 1, 3                 ; #Interfaces, Configuration#, Config. Name
        DB      10000000b               ; Attributes = Bus Powered
        DB      50                      ; Max. Power is 50x2 = 100mA
RAWInterfaceDescriptor:
        DB      9, 4                    ; Length, Type
        DB      0, 0, 0                 ; No alternate setting, only uses EP0
        DB      3                       ; Class = Human Interface Device
        DB      0, 0                    ; Subclass and Protocol
        DB      0                       ; Interface Name
RAWHIDDescriptor:
        DB      9, 21H                  ; Length, Type
        DB      0, 1                    ; HID Class Specification compliance
        DB      0                       ; Country localization (=none)
        DB      1                       ; Number of descriptors to follow
        DB      22H                     ; And it's a Report descriptor
        DB      LOW(RAWReportLength), HIGH(RAWReportLength)
```

```
RAWEndpointDescriptor:
        DB      7, 5                    ; Length, Type
        DB      10000001b               ; Address = IN 1
        DB      00000011b               ; Interrupt
        DB      64, 0                   ; Maximum packet size
        DB      100                     ; Poll every 0.1 seconds
RAWConfigLength   EQU $ - RAWConfigurationDescriptor
RAWReportDescriptor:                    ; Generated with HID Tool, copied to here
        DB      6, 0, 0FFH              ; Usage_Page (Vendor Defined)
        DB      9, 1                    ; Usage (I/O Device)
        DB      0A1H, 1                 ; Collection (Application)
        DB      19H, 1                  ;   Usage_Minimum (LED 1)
        DB      29H, 8                  ;   Usage_Maximum (LED 8)
        DB      15H, 0                  ;   Logical_Minimum (0)
        DB      25H, 1                  ;   Logical_Maximum (1)
        DB      75H, 8                  ;   Report_Size (8)
        DB      95H, 250                ;   Report_Count (250)
        DB      81H, 2                  ;   Input (Data,Var,Abs)
        DB      91H, 2                  ;   Output (Data,Var,Abs)
        DB      0C0H                    ; End_Collection
RAWReportLength   EQU $-RAWReportDescriptor

ASCIIConfigurationDescriptor:
        DB      9, 2                    ; Length, Type
        DB      LOW(ASCIIConfigLength), HIGH(ASCIIConfigLength)
        DB      1, 1, 4                 ; #Interfaces, Configuration#, Config. Name
        DB      10000000b               ; Attributes = Bus Powered
        DB      50                      ; Max. Power is 50x2 = 100mA
ASCIIInterfaceDescriptor:
        DB      9, 4                    ; Length, Type
        DB      0, 0, 0                 ; No alternate setting, only uses EP0
        DB      3                       ; Class = Human Interface Device
        DB      0, 0                    ; Subclass and Protocol
        DB      0                       ; Interface Name
ASCIIHIDDescriptor:
        DB      9, 21H                  ; Length, Type
        DB      0, 1                    ; HID Class Specification compliance
        DB      0                       ; Country localization (=none)
        DB      1                       ; Number of descriptors to follow
        DB      22H                     ; And it's a Report descriptor
        DB      LOW(ASCIIReportLength), HIGH(ASCIIReportLength)
ASCIIEndpointDescriptor:
        DB      7, 5                    ; Length, Type
        DB      10000001b               ; Address = IN 1
        DB      00000011b               ; Interrupt
        DB      64, 0                   ; Maximum packet size
        DB      100                     ; Poll every 0.1 seconds
ASCIIConfigLength  EQU $ - ASCIIConfigurationDescriptor
ASCIIReportDescriptor:                  ; Generated with HID Tool, copied to here
        DB      6, 0, 0FFH              ; Usage_Page (Vendor Defined)
        DB      9, 1                    ; Usage (I/O Device)
        DB      0A1H, 1                 ; Collection (Application)
        DB      19H, 1                  ;   Usage_Minimum (LED 1)
        DB      29H, 8                  ;   Usage_Maximum (LED 8)
        DB      15H, 0                  ;   Logical_Minimum (0)
        DB      25H, 1                  ;   Logical_Maximum (1)
        DB      75H, 8                  ;   Report_Size (8)
        DB      95H, 250                ;   Report_Count (250)
        DB      81H, 2                  ;   Input (Data,Var,Abs) = bits
```

```
        DB      95H, 40                 ;  Report_Count (40)
        DB      91H, 2                  ;  Output (Data,Var,Abs) = characters
        DB      0C0H                    ; End_Collection
ASCIIReportLength  EQU $-ASCIIReportDescriptor
String0:                                ; Declare the UNICODE strings
        DB      4, 3, 9, 4              ; Only English language strings supported
String1:                                ; Manufacturer
        DB      (String2-String1),3     ; Length, Type
        DB      "U",0,"S",0,"B",0," ",0,"D",0,"e",0,"s",0,"i",0,"g",0,"n",0," ", 0
        DB      "B",0,"y",0," ",0,"E",0,"x",0,"a",0,"m",0,"p",0,"l",0,"e",0
String2:                                ; Product Name
        DB      (String3-String2),3
        DB      "R",0,"e",0,"a",0,"d",0,"e",0,"r",0," ",0
        DB      "B",0,"o",0,"a",0,"r",0,"d",0
String3:                                ; Configuration 1 Name
        DB      (String4-String3),3
        DB      "R",0,"A",0,"W",0," ",0,"D",0,"a",0,"t",0,"a",0
String4:                                ; Configuration 2 Name
        DB      (EndOfDescriptors-String4),3
        DB      "A",0,"S",0,"C",0,"I",0,"I",0," ",0,"D",0,"a",0,"t",0,"a",0
EndOfDescriptors:
        DW      0                       ; Backstop for String Descriptors
```

Figure 6-25. Supporting two configurations

Example 4—Step 3: Implement Microcontroller Code

From the microcontroller's perspective, the difference in the configurations is how the data buffer received from the host PC is interpreted. No interpretation is done in the **raw** configuration, and character lookup is done in the **ASCII** configuration. This is a simple example of multiple configurations, so you can focus on the design method.

The microcontroller code for implementing this two-configuration example is included on the CD-ROM for detailed review.

Example 4—Step 4: Application Code

The application program is also enhanced to support the two configurations. In ASCII mode the entered characters are set to the I/O device, which does the character lookup and then returns a raw buffer for editing if required. The source code for this Visual Basic application is also included on the CD-ROM for you to edit and enhance.

CHAPTER SUMMARY

We accomplished a great deal in this chapter. We implemented the concepts we learned from previous chapters into a working "buttons and lights" example. This simple example demonstrated the PC host manipulating a physical external device (the LEDs) and demonstrated a physical external device (the BUTTONS) controlling the operation of the PC host. This milestone was made possible by the underlying infrastructure of USB.

The basic design was then extended to access more physical I/O. An I/O device with two configurations was built, and an application program manipulated both configurations.

We will build on this knowledge in the upcoming chapters. If we can operate some buttons and lights, we ought to be able to operate *anything*. . . .

Chapter 7
Migration from ISA

So what's wrong with ISA? Why am I encouraging you to move off of the ISA bus? It boils down to **system performance**—have you noticed how your Internet access slows down while you are printing, how your screen seems to freeze when scanning or accessing a floppy disk? These effects are due to the operation of the ISA bus.

If we compare the architecture of the first PC (Figure 7-1) with that of a modern PC (Figure 7-2), we notice that both have a processor at the top and the ISA bus at the bottom. The original PC contained an 8-MHz Intel 8088 processor; in many PCs today, the processor is a 400-MHz Intel Pentium II processor. Processor performance has increased by over a thousand times while the ISA bus has remained the same.

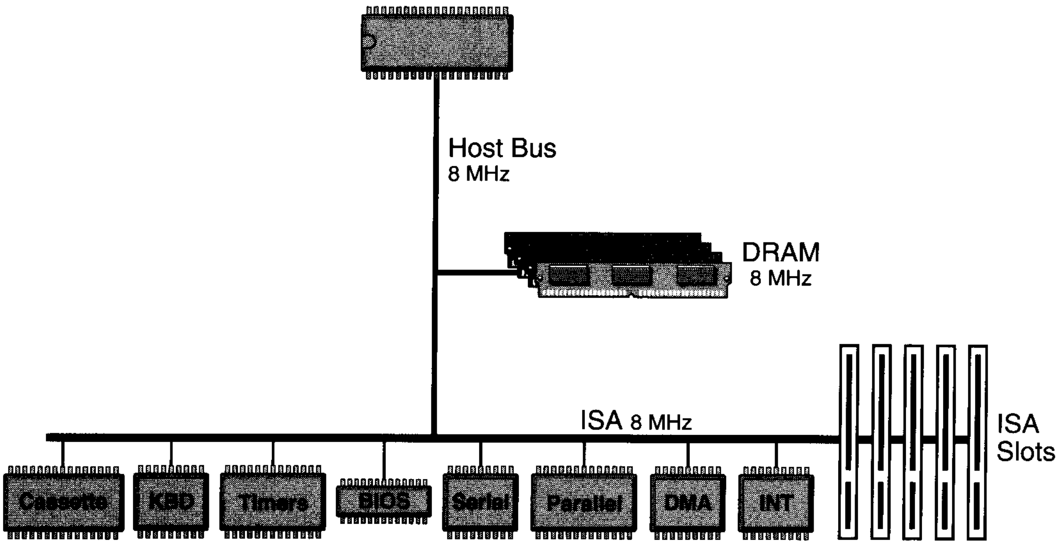


Figure 7-1. Original IBM PC architecture

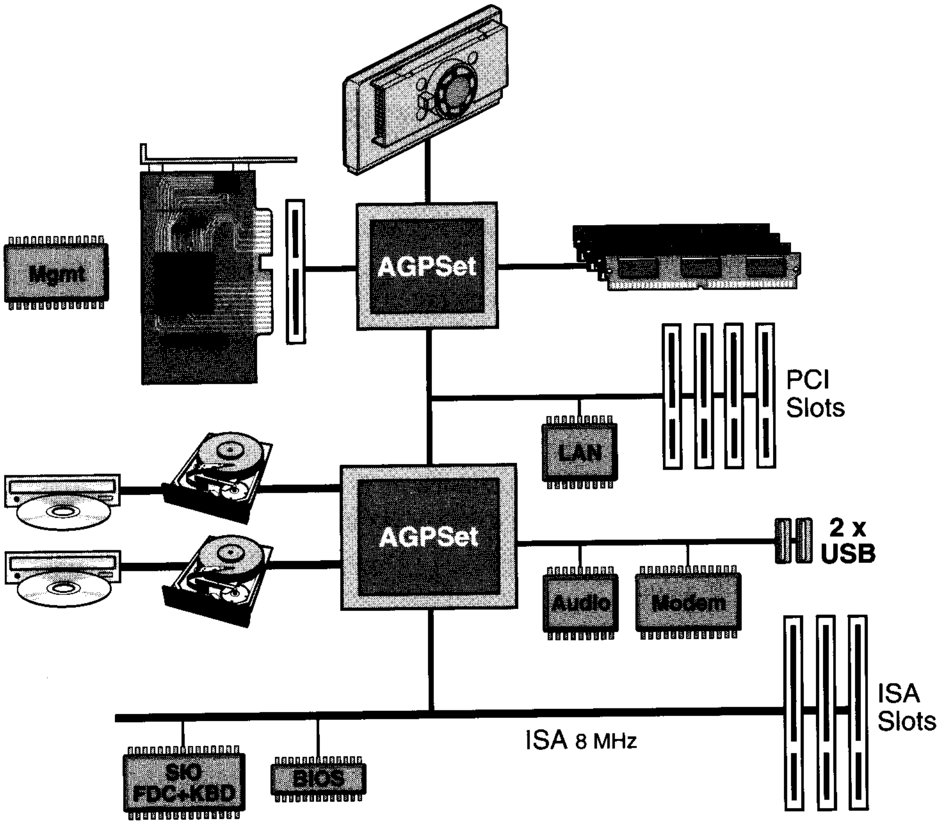


Figure 7-2. Today's PC architecture

Figure 7-3 shows another reason to migrate away from ISA: Tomorrow's PC will not have an ISA bus! The PC 98 and PC 99 System Design Guide documents from Intel and Microsoft have predicted the end of ISA since the end of 1997. PC platforms with ISA-based peripherals will not receive Microsoft's "Designed for Windows" Logo from June 1999 on.

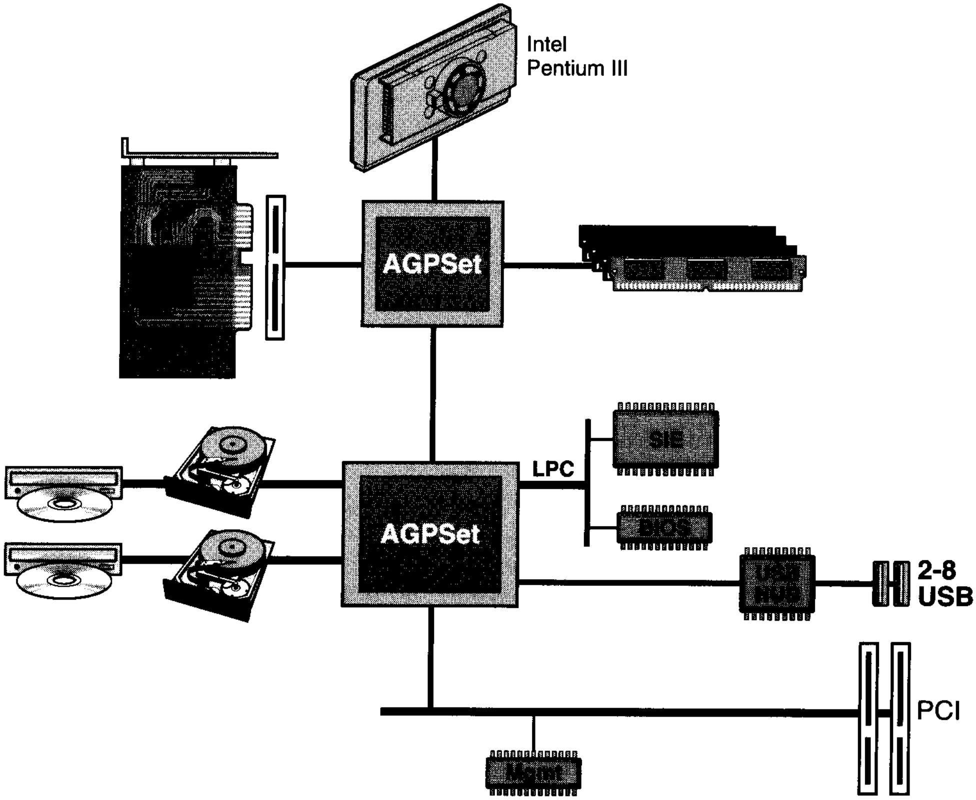


Figure 7-3. Tomorrow's PC architecture

The ISA bus limits growth of the PC platform because of its operation with today's processors and its general lack of modern features such as power management, hot-swap, and configuration management.

IN AND OUT ARE SPECIAL

Fundamental to Intel Architecture microprocessors is the mechanism to access real-world Input/Output. The Intel Architecture instruction set includes specific instructions, IN and OUT, to manipulate data from the real world. Because the timing of a real-world device is not known, the processor waits for an IN or OUT instruction to complete before moving on. In the "old" PC with the 8088 processor, this effect of waiting for completion could be ignored because instruction timings and I/O timings were about the same. Today, if we want optimum system performance, we need to take the IN/OUT semantics into account. The Pentium II processor gains performance by dynamically reordering the instructions it is executing; it also looks ahead in the instruction stream to determine the optimal order in which to execute instructions. The processor will not reorder IN and OUT instructions, because they are tied to events in the real world. While waiting for an IN or OUT instruction to complete, the processor loses the opportunity to optimize program flow.

The real loss in performance is caused by the slow ISA bus. Referring to today's architecture (Figure 7-2), when the processor issues an IN or OUT instruction, it must gain and hold the PCI bus. Once the processor owns the PCI bus, it must then gain and hold the ISA bus; the instruction is completed, and all the buses are now freed. Unfortunately, ISA timing matches the timing of an 8-MHz 8088 processor; so, in the same time period, a 400-MHz Intel Pentium II processor could have executed 400 to 600 instructions. Retaining the ISA bus in a system costs us the opportunity to increase system performance.

Building an ISA Card

Because there was initially no official documentation on how to build an ISA card, many people "reverse-engineered" the requirements from IBM's original design. This worked for a while, until other manufacturers such as Compaq started to build IBM-compatible PCs. The lack of ISA documentation caused minor incompatibilities and resulted in some ISA cards not operating in some machines.

Intel processors support 64 KB of Input/Output address space, but the original PC design included only 10 of the 16 I/O address lines. This resulted in an I/O space of only 1 KB. IBM used some addresses in this space for the system timer, DMA, interrupt controller, and peripherals (serial ports, parallel ports, keyboard, floppy drive, speakers). The I/O addresses were extracted from the BIOS listing by each ISA card builder who located an ISA card in an unused (by IBM) address space. There were, of course, address conflicts; many manufacturers resolved this by making the ISA card address jumper-selectable. But this simply moved the problem to the user.

ISA card manufacturers also used the IBM system interrupts and DMA channels. These too had to be made jumper-selectable, adding confusion for the user. These limited resources were soon used up, and it became burdensome to add more ISA cards to a PC platform.

The inadequacies of ISA were highlighted when the PC industry first started to ship "multimedia upgrade kits." It was difficult to add sound to an early PC, and it required an ample supply of system resources such as interrupt lines and DMA channels. The user was initially reluctant to take the cover off the PC. Would you take the cover off your VCR to "upgrade" it? Tiny jumpers had to be set, ISA cards had to be reinserted into the PC platform, and software had to be installed. It shouldn't surprise us that over 60 percent of these upgrade kits were returned! The upgrade process was too difficult and error-prone for the average consumer to deal with. This was a very valuable (although expensive) lesson for the PC industry. The PC platform had been embraced by the engineering community, but the consumer is a very different customer—to whom even the color and shape of the box are part of the purchase-decision process. Of **paramount** importance is **Ease of Use**.

Plug and Play ISA

To try to make it easy for a user to add ISA cards to a PC, Microsoft launched a Plug and Play campaign within the ISA card manufacturer community. Many electronics manufacturers collaborated with Microsoft to define mechanisms and standards that would enable ISA cards to be programmable. The operating system would then interrogate the new ISA cards to understand their system resource requirements and to program each card's I/O address, interrupt, and DMA channel to be exclusive.

Unfortunately, the hardware complexity of the resulting Plug and Play implementation far exceeded the functionality of the simpler I/O cards. Manufacturers of the simpler cards ignored the recommendations. Manufacturers of high-functionality cards integrated Plug and Play capability into their bus interface ASICs, so the general ISA situation was improved—but the situation was not ideal.

A modern Plug and Play requirement is power management. With power management, a device powers itself down if it is not being used. The ISA bus cannot support this; in fact, an ISA card can prevent a PC from powering down even when the ISA card is not being used. The ISA bus cannot support "hot-plug" either (hot-plug means that cards are inserted and extracted with the system power still applied). This general lack of features has caused many people to migrate off of ISA already. But you must still have an ISA-based design, because you are reading this chapter!

To increase the throughput of an ISA system, Compaq Computer proposed and drove a standard extension to ISA called EISA—this extension stretched the bus to 32 bits and added block mode transfers. The Plug and Play scheme for ISA was also embraced and extended. Best of all, however, was a two-tier connector that allowed standard ISA cards to plug in and operate as before while EISA cards would connect to the added signals and thus operate with increased functionality. This extension served the industry well for many years but has now been almost completely replaced by PCI.

MIGRATION FROM ISA

The discussion so far has looked at half of the migration task—the hardware. The software is equally important, and for this chapter I assume that the original application code cannot be changed. That is, the software designer may have moved on or the original source code may have been lost. So our solution will be based on the executable object code. Figure 7-4 shows our current design task. We have a software application that makes MS-DOS calls and directly accesses some custom hardware. MS-DOS, of course, also accesses the hardware. The hardware is an ISA card with custom I/O that interfaces to the ISA bus.

We'll look at the software and hardware tasks separately in the discussion that follows.

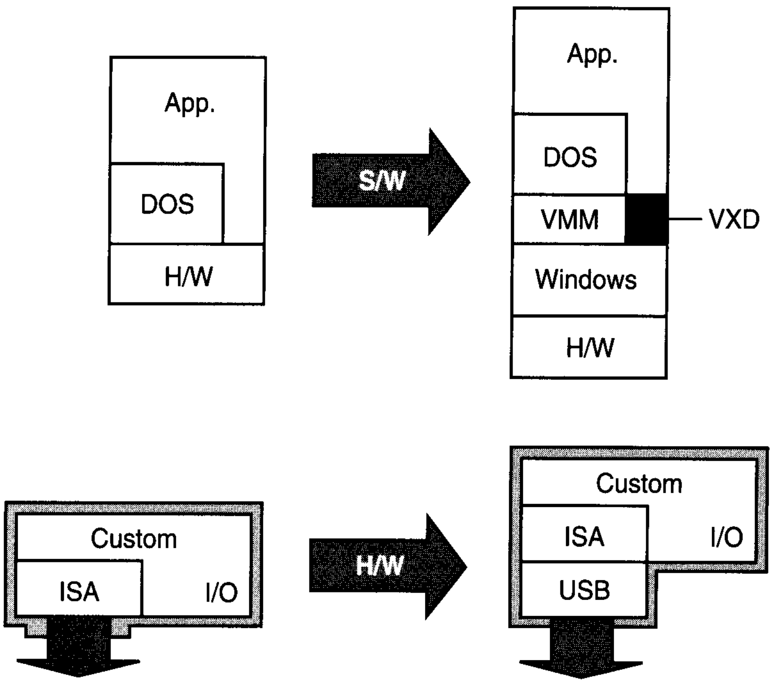


Figure 7-4. The migration design task

Software Migration

I assume the application program is written for the MS-DOS environment, because this has been the case with most customers I have discussed this solution with. Our goal is to move this program, unchanged, to a Windows environment. By unchanged, I mean that we are going to move the executable version of the program. I assume we do not have the luxury of being able to edit the source code. This approach will create a general solution for all MS-DOS programs and not a specific solution for a single application program.

In Windows, Microsoft included many features to allow this OS to easily support MS-DOS programs. Microsoft's focus was on general applications, games, and software that directly manipulate the I/O devices found on a typical PC (for example, mouse, gameport, SoundBlaster sound, printer, and serial ports). They did a very thorough job and supplied many pieces of system software that support almost every commercial MS-DOS application software that can be purchased. We'll look at how they did this task, and we'll use the same techniques to move our custom, ISA hardware–based MS-DOS application program into a Windows environment.

MS-DOS View of the World

When writing an MS-DOS program, programmers assume they have total and singular control over the hardware inside the PC. Some memory is reserved for MS-DOS and some system I/O devices are also reserved, but in general all of the microprocessor's address space and all hardware components inside the PC are available for programmers to use. The excellent documentation on the inner construction of PC hardware provided by IBM also includes a BIOS listing so MS-DOS programmers could get very close to the hardware devices.

Our MS-DOS program talks to custom I/O on an ISA card. The first design decision is where to put these I/O ports in the PC I/O address map. The IBM PC design chose I/O port addresses for all of the system hardware, and lists were published of these reserved addresses (I have included such a list in the Chapter 7 directory on the CD-ROM). Some unused I/O addresses were chosen, and the custom I/O card was designed around these. For this example, let's assume that I/O addresses from 300H to 30FH are being used.

Windows View of the World

Windows 98 is a 32-bit, multitasking operating system that can support several application programs running at the same time. The programs could be MS-DOS–based, and, as we have just learned, each program believes it has exclusive use of all of the PC host hardware. How can this work out? Let's look a little deeper into **how** Windows 98 is able to run multiple programs, and we'll discover some excellent techniques that will let us easily move our custom I/O application program.

Windows 98 makes extensive use of the protected-mode features of the Intel Architecture processor. From the Intel486 processor on, all Intel Architecture processors have supported a four-ring protection model (Figure 7-5). In a Windows environment, the operating system is in the central, most privileged ring, and application or user programs are in the outermost, least privileged ring.

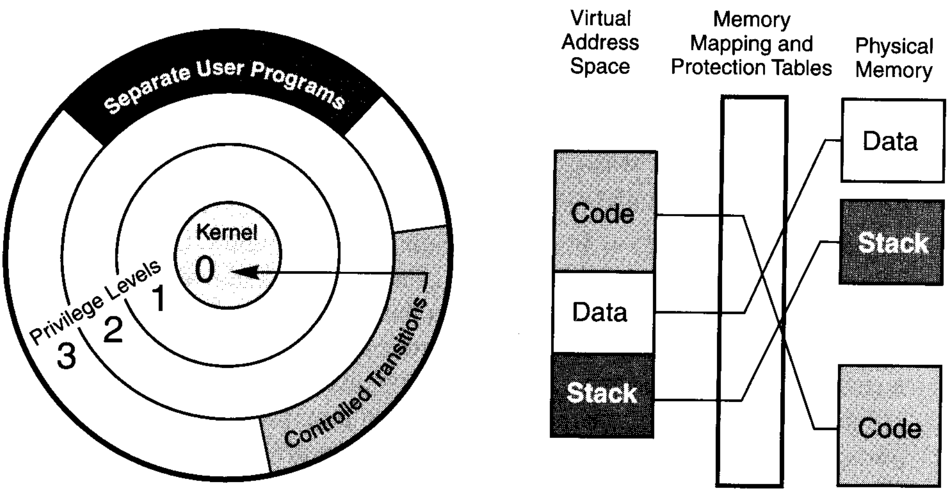


Figure 7-5. Intel Architecture protection model

During the initialization of protected mode, the operating system programmer can choose to disallow certain instructions to be executed in ring 3. Key instructions such as "Disable Interrupts" or "Halt" would affect the reliability of the operating system and are therefore not permitted to be executed from a user program. If you try to execute them from a user program, the hardware protection mechanisms inside the Intel processor will trap these privileged instructions **before** they execute and will vector the processor to an interrupt service routine inside ring 0. The operating system then decides what to do with the "buggy" application program; choices can include terminating the program and deleting it from memory!

User programs also operate in a virtual address space. When writing an application, the MS-DOS programmer assumes access to all of the processor's memory space. When the Windows operating system loads a program into memory, the OS needs the flexibility to load the program into any free physical memory space. During loading, the OS creates a system memory-mapping table that the user program accesses transparently to create a physical address. This table includes **limits,** so a "buggy" application program that tries to access a memory location outside the range that has been allocated will cause a protection trap to the operating system—which can deal with this "memory violation error." This scheme prevents multiple application programs from interfering with each other and with the operating system.

Of particular interest to us for this example are the IN and OUT instructions. These are privileged instructions that a user program is not allowed to execute. Attempts to execute them cause a trap to the operating system. Because Microsoft expected many programs to continue to include IN and OUT instructions, their solution to this "illegal instruction" is very elegant and efficient. Let us first agree that an application program cannot own any real I/O ports—this would be disastrous if two running programs were to intermingle I/O requests to a printer, for example. All the real I/O ports are owned by the operating system, which can selectively allow programs to believe they are accessing I/O ports. The terms "hardware emulation" and "Virtual Device Driver" are used to describe this implementation.

The Windows 98 operating system has the capability to install a virtual device driver, or V*x*D, on a per I/O port basis. Typically, however, a V*x*D will service a group of I/O ports. The operating system includes a complete set of V*x*Ds that manage user access to system I/O resources such as the serial ports and printer ports. So a "simple" IN or OUT instruction, when used in a user application program, actually causes the operating system to execute a virtual device driver that does the real I/O for us. All this is transparent to the user program except that the instruction can take many hundreds of system clocks rather than four. We will demonstrate this timing difference in an example later in this chapter.

I have included a utility on the CD-ROM, in the Tools directory, called VXDVIEW. This program was written by Vireo Software and is supplied with their permission. VXDVIEW allows you to observe the V*x*Ds currently running on your PC host—there are a lot of them, and we will add a new one later in this chapter.

So, in summary, each application program has a protective "wrapper" around it so that the operating system can monitor and control its execution. This allows multiple applications to run at the same time, each believing that it has exclusive access to the PC system.

OUR CUSTOM I/O EXAMPLE

Now that we understand the Windows mechanisms for supporting memory and I/O accesses from a user program, we can implement a solution for our custom I/O board. Remember that our example application program makes I/O accesses to ports 300H to 30FH. Figure 7-6 shows our solution diagrammatically.

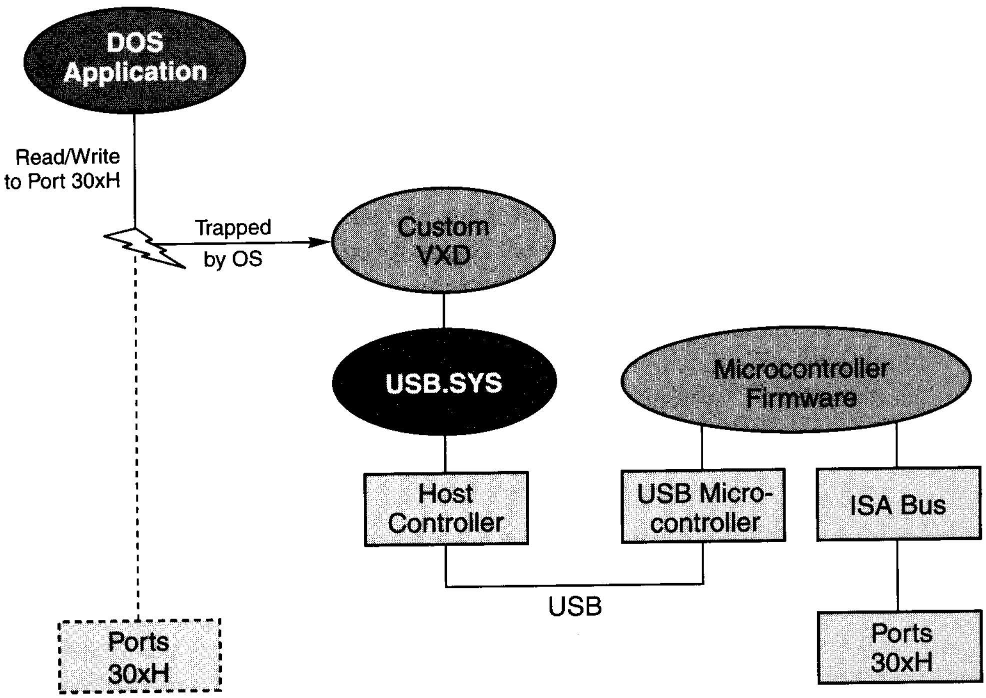


Figure 7-6. Implementing a "hardware redirector"

We'll write a virtual device driver that is called when the user program wants an I/O access completed, and we'll send the request to a USB microcontroller. The microcontroller will implement the real I/O access and pass the results back to the virtual device driver for a read. In turn, the driver will return to the user program. Thus, we are transparently redirecting the I/O accesses—the user program will not know that this is happening "behind-the-scenes" and need not be modified in any way. This sounds as if it will take a LONG time to execute, but in fact it is similar to other system V*x*Ds in operation. We shall measure the actual performance of our solution later.

To make this example more tangible, consider the simple ISA I/O board shown in Figure 7-7. This board is a custom display system made up of eight components, each of which has a matrix of 5x7 LEDs in it. The total LED matrix is 40 LED columns long by 7 LEDs high. The ISA card strobes the columns on, one at a time, and relies on human persistence of vision to see all of the display actually lit up. A shift register arrangement is used to strobe each column on in turn. The board has four I/O addresses that are configurable at four different ISA base addresses. ISA I/O writes are used to update the display.

The MS-DOS application program reads up to six characters from the command line and then writes to the display. The program does the character lookup for the 5x7 character to create a 40-byte buffer that is then sent to the custom I/O board. This buffer is continually sent to the display to keep it refreshed, and, for demonstration purposes, all of the operations can be timed.

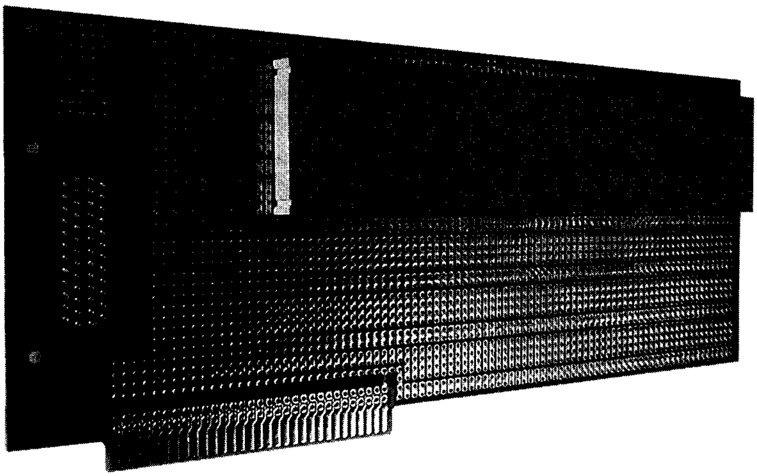
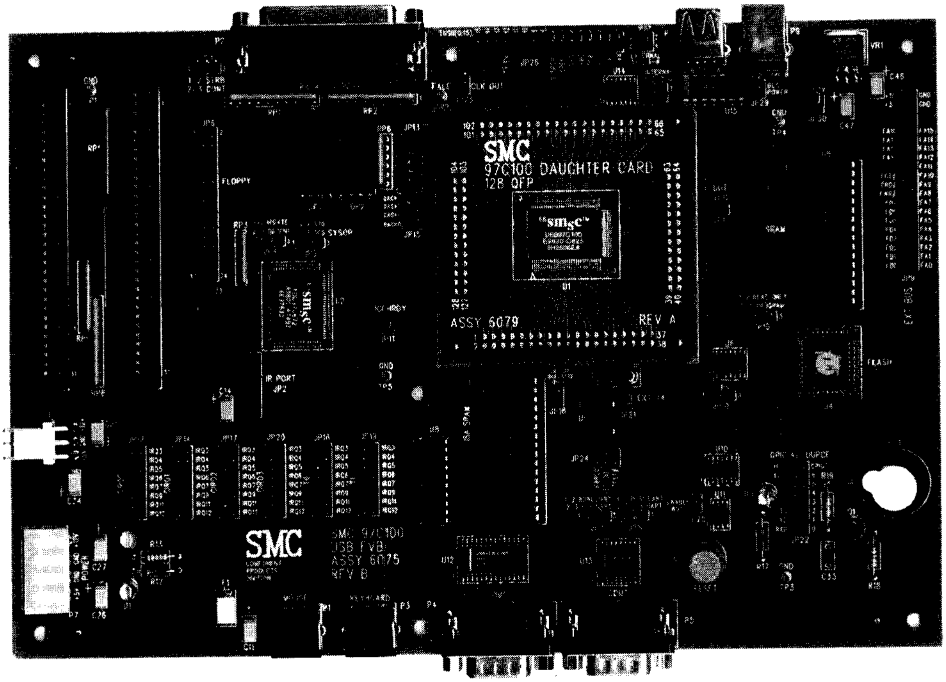


Figure 7-7. Custom ISA display board

Figure 7-8 shows a development board from SMSC Microsystems. SMSC has been a major supplier to the PC motherboard market, and their Super I/O components appear in many production designs. They understand the PC-style I/O business and are excited about the new PC I/O possibilities that USB is creating.



Courtesy of Standard Microsystems Corp.

Figure 7-8. USB+ISA development board from SMSC

The development board shown in Figure 7-8 is a USB-based peripheral that has a full complement of PC-style I/O (parallel port, serial ports with infrared, keyboard, mouse, floppy disk controller, speaker) and two ISA slots. In operation it connects a complete PC I/O subsystem at the end of a USB cable. USB is fast enough to support this I/O subsystem at full speed.

SMSC designed and manufactures a special USB microcontroller to excel at ISA I/O device operation. SMSC's expertise and knowledge of ISA systems was used to create the 97C100 controller whose block diagram is shown in Figure 7-9. The 97C100 uses an MCS51 microcontroller as a protocol engine and has a wide array of special-purpose hardware to support the ISA bus in all of its operating modes. The 97C100 can maintain ISA timing, and an ISA add-in card will not know that it is not inside a PC.

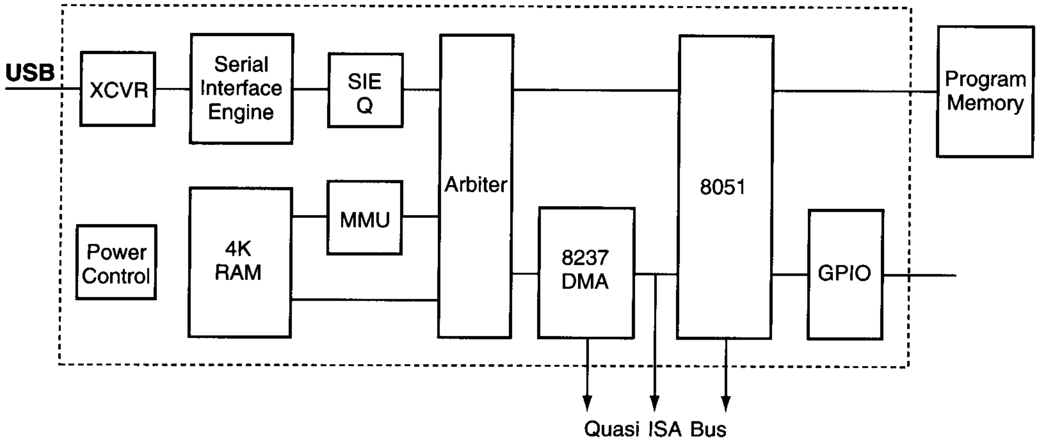


Figure 7-9. Block diagram of 97C100

Our custom I/O board uses only programmed I/O and is very easy for the SMSC development board to support. SMSC has demonstrated more complex cards, using ISA interrupts and DMA, operating successfully in this environment. I have used a simple example so we can study the concepts.

So the first step in our migration is to remove the custom ISA card from the PC and insert it into an ISA slot on the SMSC development board (Figure 7-10).

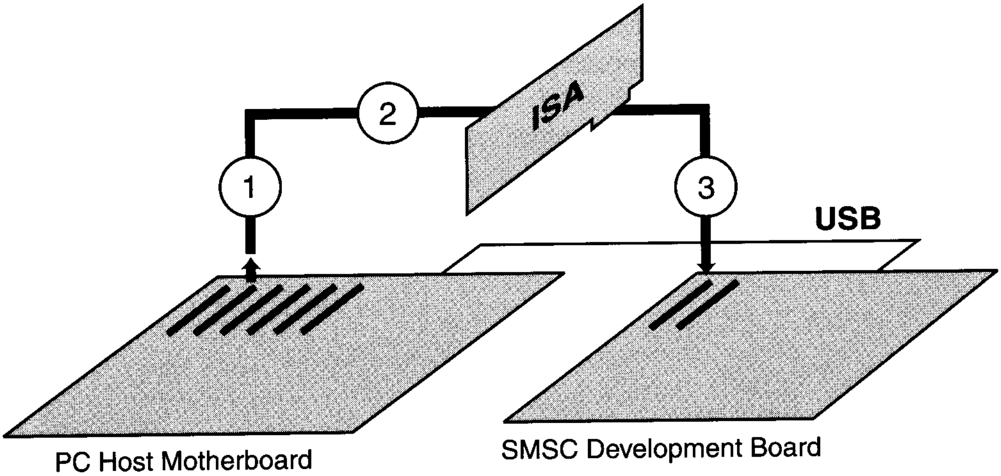


Figure 7-10. Moving the ISA card

The V*x*D is now loaded onto the PC and invoked. The V*x*D installs trap vectors for I/O ports 300H to 30FH. Any read or write to this range of ports will cause a USB interrupt transfer to be generated and sent to the 97C100 board, just like all the examples in the previous chapter. On receiving the interrupt transfer, the 97C100 creates an ISA write cycle that updates our display. The V*x*D and the source of the 97C100 code can be found in the Chapter 7 directory of the CD-ROM.

The first stage of the migration is complete! We have successfully moved a custom ISA card out of the PC and into a USB device without having to change the software that was driving the I/O card. Measuring the performance of this simple I/O card using system timers showed no detectable difference (less than 5 percent difference).

Notice that we have accomplished a huge side benefit—we have decoupled the custom MS-DOS software from the custom ISA hardware. The original IN and OUT instructions are still there, but we have intercepted them using a virtual device driver and have redirected them to new hardware. If we keep the same interface to the MS-DOS application, we can change the hardware to achieve lower costs if desired.

The second stage of the migration would be optimization of the hardware solution.

Design Optimizations

The actual design directions taken at this point depend on the complexity of your real MS-DOS/ISA solution and how many units you need to build (balancing cost and time targets). I'll describe basic directions to let you choose the one best for your project.

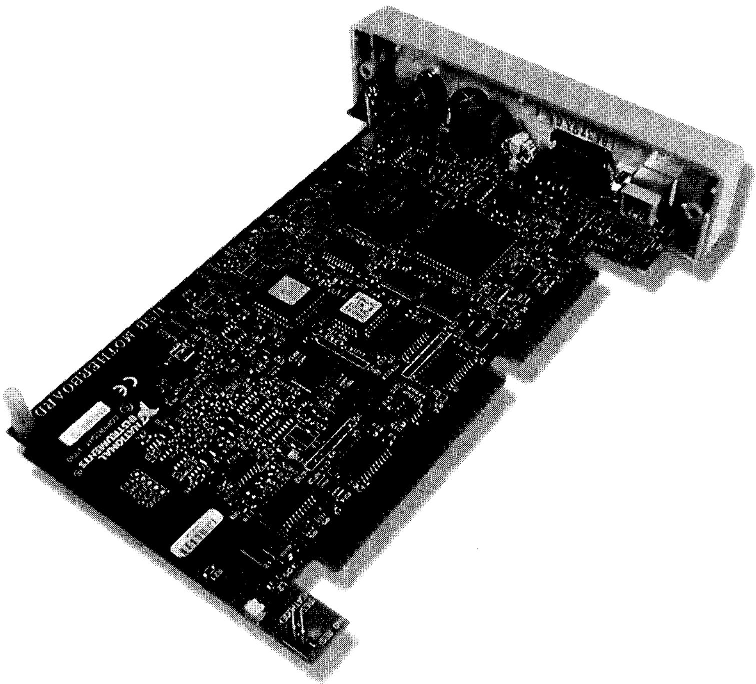
Multiboard Example

If your custom ISA I/O consists of several boards, it would be less risky to optimize the design of the SMSC development board. The key component, 97C100, can drive up to four ISA slots, so a range of solutions is possible. The development board includes schematics in ORCAD format, so the design challenge is reduced to a baseboard layout exercise—good low risk.

I do **not** advocate this ISA-based I/O subsystem rack for new designs, because there are much better methods for solving this problem. What I have described is a **transition** solution.

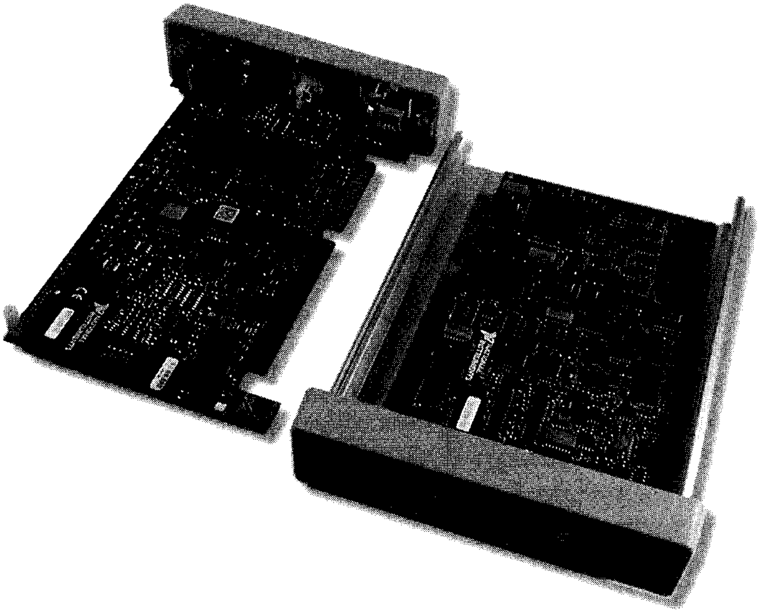
Many Individual Boards Example

A very creative solution implemented by National Instruments may apply to your situation. National Instruments has a wide range of individual ISA boards that implement interfaces to analog and digital instrumentation. A custom USB-to-ISA bridge board, shown in Figure 7-11, enabled National Instruments to create an "instant" USB product line. The bridge board, and its companion ISA board, are powered from USB; this allows the pair of boards to be packaged in a convenient, stand-alone enclosure as shown in Figure 7-12.



Courtesy of National Instruments Corp.

Figure 7-11. A custom USB-to-ISA bridge



Courtesy of National Instruments Corp.

Figure 7-12. Creative packaging produces a USB I/O device

Single, Simple ISA Board Example

If the custom ISA card is simple, as it was in this display example, with no interrupts and no DMA, then we don't really need the elegance and thoroughness of the 97C100 solution. A simpler USB microcontroller could be used (Figure 7-13).

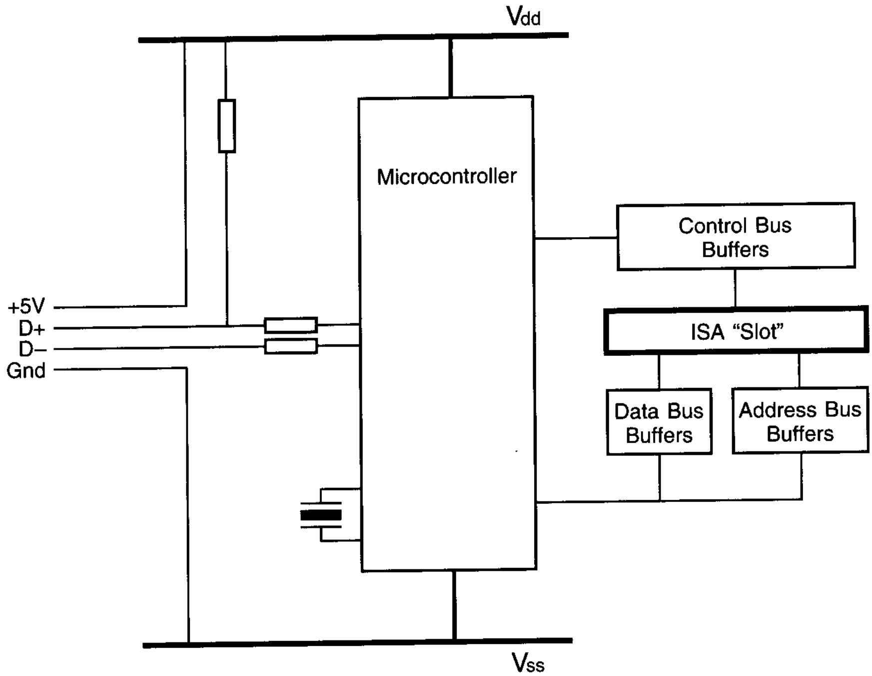


Figure 7-13. Simple ISA card migration

The USB microcontroller in Figure 7-13 re-creates an ISA bus using its I/O ports. The ISA address is generated, and then ISA controls are generated under program control of the microcontroller. The bus timings will **not** be the same as ISA timings (as they were with the 97C100 example), so this introduces some risk. For simple I/O boards, however, such as our custom display example, the ISA signal timing was not a critical part of the design. Example microcontroller code for a simple ISA bus converter can be found in the Chapter 7/Anchor Chips directory on the CD-ROM.

Depending on your packaging or other hardware constraints, the ISA interface on your custom I/O board could be replaced with a USB interface. If the total power drawn by the custom I/O is less than 500 mA, then we can even supply the power from USB. We have just built a USB dongle that operates the same as the original ISA-based design, and we didn't have to change the MS-DOS software at all! (A "dongle" is an intelligent adapter. A physical adapter lets you connect, for example, a serial mouse to a PS/2 port. But a dongle also changes signal levels and protocols, etc.)

CHAPTER SUMMARY

Migrating from an MS-DOS/ISA environment into a Windows/USB environment is now straightforward if you use the tools and techniques described in this chapter. No change to the original MS-DOS application software is required—our solution creates a software-transparent link. Furthermore, the solution decouples the ISA hardware from the MS-DOS software so we can consider hardware design optimizations. The I/O solutions can range from a complete slave ISA-based I/O subsystem, through a single board ISA bridge, to a simple, optimized microcontroller solution. The flexibility of USB also lets you choose a solution that best matches the application that **you** are moving.

Chapter 8
Building USB Bridges

In this chapter we assume that you already have a PC peripheral, such as a modem, scanner, or other device that attaches to the PC's RS-232 serial or parallel ports. Your peripheral works fine, but you're considering a USB connection because of the excitement around USB and the reality that these "legacy" ports will be eliminated on future PCs.

We'll present multiple alternatives so you can choose the one that's best for your product.

- First we'll design a USB-to-serial bridge. We've supplied code for this alternative.
- Then we'll design a high-speed communications peripheral—a 56-Kbps modem—that uses a USB connection.
- We'll look at other standard connections found in the PC industry, such as floppy disk, SCSI, and CardBus, and build bridges from all of them to USB.
- And we'll finish with a design that converts a custom peripheral, a bar code scanner, into a USB peripheral.

Adding USB capability to an existing product is straightforward, and the new product inherits all of the benefits of USB.

Design of a USB–to–RS-232 Bridge

The RS-232 connection to a PC has proved to be a popular interface choice for peripheral device manufacturers—so popular, in fact, that many users have had to buy more plug-in serial port cards or external switch boxes to accommodate all of the peripherals! The original IBM PC had a single serial port and allowed adding three more. The MS-DOS operating system called these ports COM1, COM2, COM3, and COM4, and most Windows applications also assume that up to four devices exist.

One drawback of the RS-232 connection is that it is point-to-point, which means that only one peripheral device can be connected to one serial port. Multidrop serial port connections, such as RS-422, exist in the industry but have not proved popular in the PC peripheral business because no PC system manufacturer has put an RS-422 connector on their motherboard.

In contrast, USB can be thought of as a multidrop, high-speed connection. Note that USB is NOT a multidrop serial bus like RS-422; it is point-to-point, and hubs are used to enable more connectivity. But a USB solution operates as, and has similar attributes to, a multidrop solution.

Figure 8-1 shows an existing PC peripheral connected to a PC host using an RS-232 connection.

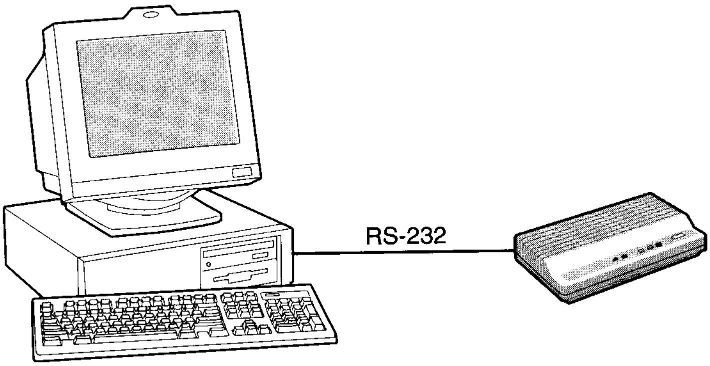


Figure 8-1. Using an RS-232 connection between PC and a peripheral

Using our buttons-and-lights example from Chapter 6, we swap the buttons for an RS-232 receiver and swap the lights for an RS-232 transmitter. Then we modify the firmware to match, and we have thus built a USB–to–RS-232 bridge, or dongle (Figure 8-2).

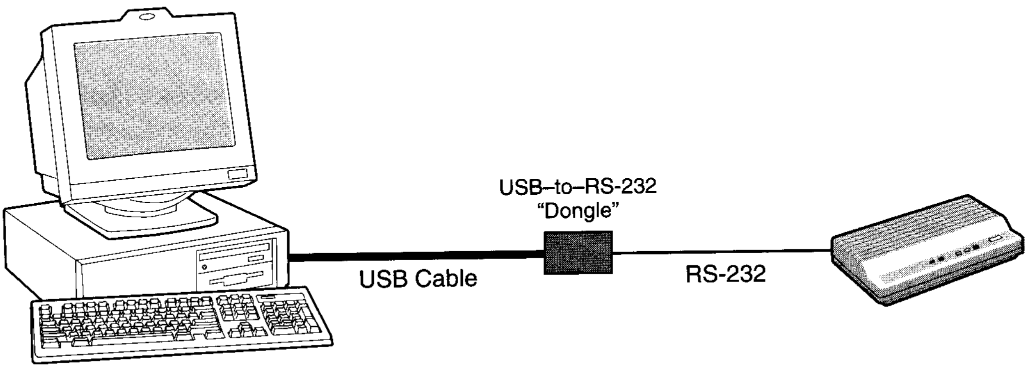


Figure 8-2. Using a USB–to–RS-232 dongle

All of the microcontrollers used in the Chapter 6 examples could easily do this USB–to–RS-232 bridging task. The task is simpler if the microcontroller has an integrated serial port, but the software needed to "bit-bang" a serial port is straightforward (there's an example in the Chapter 8/Serial Port directory on the CD-ROM).

If the serial I/O device has a low data rate, then this converted buttons-and-lights approach is a good solution. If the serial I/O device is a communications device, such as a modem, there's a better way of doing it, and a full USB modem design is implemented later in this chapter. Figure 8-3 shows a representative low-data-rate serial device. It is a security badge system from RFIdeas, and we'll use it as an example of an RS-232 serial device that will have a USB interface added.



Courtesy of RFIdeas, Inc.

Figure 8-3. Representative low-data-rate serial device

The RFIdeas AIR ID product is a proximity-activated identification system. You wear the badge, and the detector is attached to the PC that needs to be protected. When you walk up to the PC, you are automatically recognized and logged on. More important, however, is that when you walk away, you are automatically logged off. This simple security method can prevent unauthorized access to company data.

Looking more closely at the dongle and receiver connection, we find two microcontrollers: one in the bridge and one in the AIR ID receiver (Figure 8-4) (many similar devices would have the same block diagram).

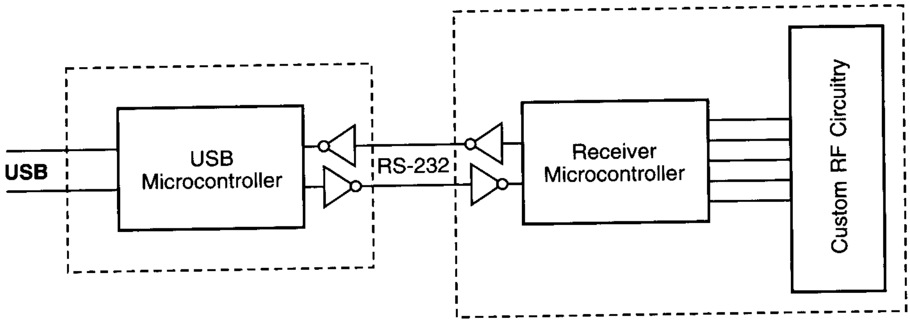


Figure 8-4. Block diagram of dongle plus serial device example

We can make some design optimizations:

- We could use a USB–to–RS-232 bridge as a separate product (several are available; see Appendix A) and offer this as an option with our serial device product. This has the lowest development cost but the highest product cost; it is useful for test-marketing a "USB-based" product.
- We could move the bridge microcontroller inside the unit (Figure 8-5). This also has a low development cost and has the flexibility of including both a serial interface and a USB interface on the product.

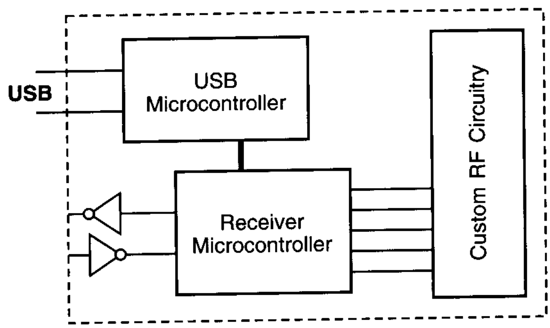
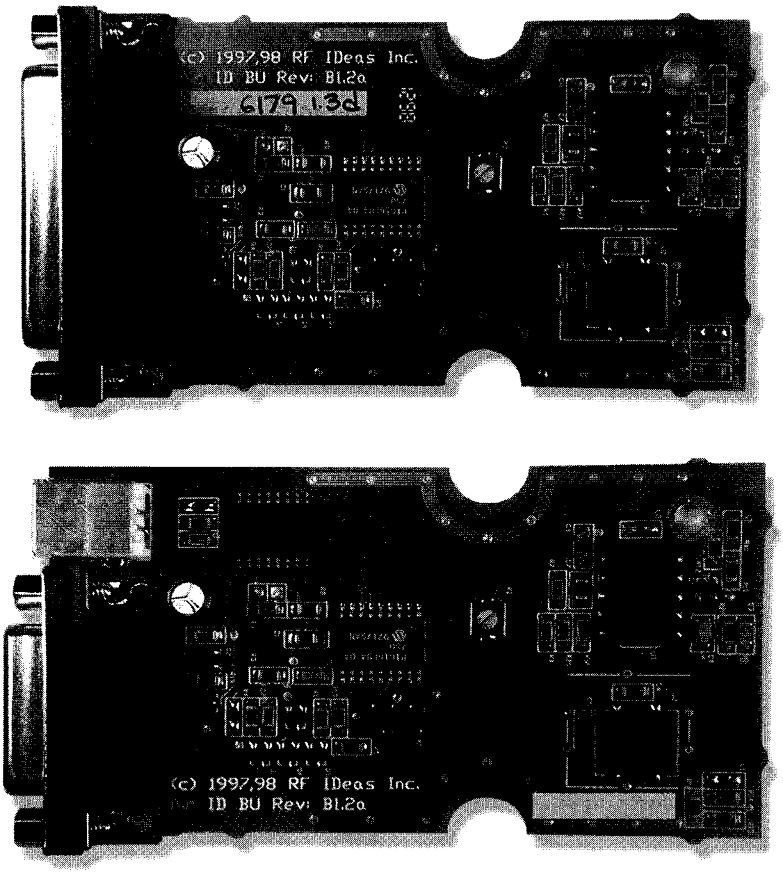


Figure 8-5. Moving the bridge inside the serial product

If our USB product proves to be very successful, we will then want to cost-reduce the dual microcontroller solution shown in Figure 8-5. The ease of this solution depends on which two microcontrollers we need to merge. If the product microcontroller is an MCS51-based microcontroller, the task is manageable, because we have included working examples with this book. If your product uses a different microcontroller, I recommend the I²C-to-USB approach. This approach involves porting the I²C example code to your microcontroller. Figure 8-6 shows an example of a completed product—it includes both a manufacturing option to build RS-232 units (as long as customers still want to buy them) and a USB option for new customers.



Courtesy of RFIdeas, Inc.

Figure 8-6. USB product built from a serial device

Now that we have a USB-connected peripheral device, we can consider adding more features. An RS-232 connection has a data channel in both directions. A USB connection has this data channel but also has a control channel to the peripheral device. Does your peripheral have option switches to set, or does it need to be calibrated, or does it need to be customized at run time? All of these features can be added at no extra hardware product cost. The USB version will be self-identifying and easier for customers to use.

The previous section focused on the RS-232 legacy connection on the PC host. The same technique can be used with parallel port devices or game port devices. The USB microcontroller is implemented as a bridge between USB and a parallel connection or a game port connection. You could select a ready-built USB–to–parallel adapter (see Appendix A), or you could integrate the device into your I/O device—similar to our serial port design.

Figure 8-7 shows a clever implementation of a game port device from Microsoft. The microcontroller inside the joystick also supports USB and feeds the D+ and D– signals to two unused pins on the game port connector. An adapter cable allows the joystick to be used in a USB environment.



Courtesy of Microsoft Corporation.

Figure 8-7. Combination game port/USB device

DESIGN OF A SERIAL COMMUNICATIONS PERIPHERAL

Our buttons-and-lights examples were implemented as HID devices so that the software task would be simpler. The HID class was designed for low-data-rate peripherals that typically interact with a person; this class does not support bulk or isochronous transfers. Interacting with another computer can be done at a much higher data rate, and the USB specification includes a Communications Class that has optimizations for this implementation. Let's use the same example of an RS-232 connected communications device, i.e., a modem, that we will migrate to a USB-connected modem.

First, a Look at Communications Standards

Let's look at the USB Communications Class in general and then investigate how USB is having a dramatic impact on the modem industry.

A great number of communications standards exist in the industry, and many of these are evolving to keep pace with new technologies that push the speed of information transfer higher and higher while bringing the cost lower and lower. The USB Communications Class does not invent new protocols or standards, nor does it dictate how existing communications equipment should be used. Rather, it defines an architecture that embraces all existing equipment but with an initial focus on telecommunications services and medium-speed networking services.

The Communications Class defines three classes:

- Communications Device Class—this is a device-level definition that identifies a communications device that has several different types of interfaces.
- Communications Interface Class—defines a general-purpose mechanism that supports all types of communication services. This is the primary interface for device management; it includes the setup and teardown of calls and management of the device's operational parameters.
- Data Interface Class—defines a general-purpose mechanism to support octet data streams using bulk or isochronous transfers. This class is used when another predefined class, such as Audio, is not appropriate.

The Communications Device Class currently outlines the following two categories of devices:

- Telecommunications devices—analog modems, ISDN (Integrated Services Digital Network) terminal adapters, digital telephones, and analog telephones
- Networking devices—ADSL modems (asymmetric digital subscriber line), cable modems, 10BASE-T Ethernet adapters, and Ethernet equivalents

A device will identify itself as a Communications Class device by entering a value of 02 as the Class code value of a device or interface descriptor. Table 8-1 shows several subclasses that are currently defined; many of these have different protocols defined.

Table 8-1. Communication interface subclass codes

Code	Subclass
0	Reserved
1	Direct Line Control Model
2	Abstract Control Model
3	Telephone Control Model
4	Multi-Channel Control Model
5	CAPI Control Model
6	Ethernet Networking Control Model
7	ATM Networking Control Model
8-127	Reserved for Future Use
128-255	Reserved for Vendor-specific Use

As seen in Table 8-1, the list of supported models is extensive. We'll work through a Direct Line Control example. Many other options are available, and the interested reader should refer to "Class Definitions for Communications Devices" in the USB Documents directory on the CD-ROM.

Direct Line Control Modem Example

A Direct Line Control modem is a recent modem implementation that has been enabled by USB. The software is simplified because of advances in modem hardware design. We'll use this hardware implementation for discussion, because it has a direct impact on the software. Figure 8-8 shows a traditional modem connected to USB using the bridge design described earlier in this chapter. Note the large collection of components: We can eliminate many of them to create a lower-cost, yet higher-performance modem.

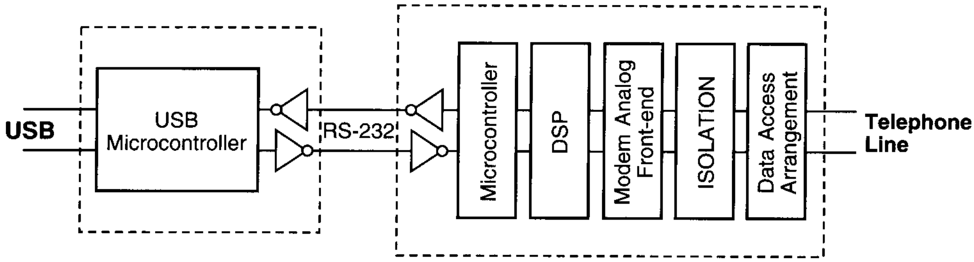


Figure 8-8. Traditional modem with USB–to–RS-232 bridge

Most traditional modems are implemented with four to six major components. These include:

- Microcontroller responsible for interpreting the Hayes modem control codes (the "AT" command set) as well as other housekeeping and speaker driver
- Digital signal processor (DSP) for signal modulation and demodulation
- Analog modulation and demodulation
- Equipment isolation
- Data access arrangement (DAA)

A modem used to be a straightforward device but today includes increasingly sophisticated modulation techniques to squeeze larger amounts of data through a fixed-bandwidth telephone line. Now, what can we start eliminating in our race to keep up with the data flow?

For example, a recently approved scheme, *ITU V.34*, defines a 1000+ point Quadrature Amplitude Modulation (QAM) trellis coding that can support 33 Kbps or 56 Kbps under certain circumstances. Designers no longer use analog methods to create these signals; the arrival of relatively inexpensive digital signal processors (DSPs) in the late 1980s changed this. The Internet has also been a driving force for change as consumers have demanded lower costs for higher speeds. USB is going to create as big a change as single-chip DSPs did.

The DAA is particular to a country and its regulations. The common thread throughout these regulations is signal isolation: The modem, or other equipment, must not be able to inject high voltages or currents into the telephone line. This isolation has been implemented using a transformer, which is bulky, heavy, and less reliable. We'll see a change here, too.

Removing the RS-232 Connection

The first thing to eliminate is the RS-232 connection. The highest speed that a PC serial port can operate at is 115.2 Kbps, and a USB connection will remove this bottleneck. The RS-232 scheme includes two serial signal paths, one in each direction, that are used to transfer in-band commands and data. Commands and status must be intermingled with data because there is only one channel in each direction. USB supports separate command and data channels, which enables out-of-band status reporting—this segregation of command and data channels enables a simplified command structure.

Moving Command Set Interpretation to the PC Host

Of the components found in a traditional modem, the microcontroller task of interpreting the Hayes command set is better handled by the PC host processor. Microsoft includes a Unimodem driver in its communications stack (Figure 8-9) to implement all of these functions. The Microsoft VCOMM port drivers can redirect output originally destined for the serial ports to be transparently passed to the Windows Device Driver Model (WDM) class driver. The WDM class driver can then direct this output to a USB device. Input from the USB device can be similarly passed back from the WDM class driver, through the VCOMM driver to the Unimodem driver, and back to the applications program. This fundamental partitioning of the logical I/O to the physical I/O that WDM provides gives Windows its "hardware-independent" feature set.

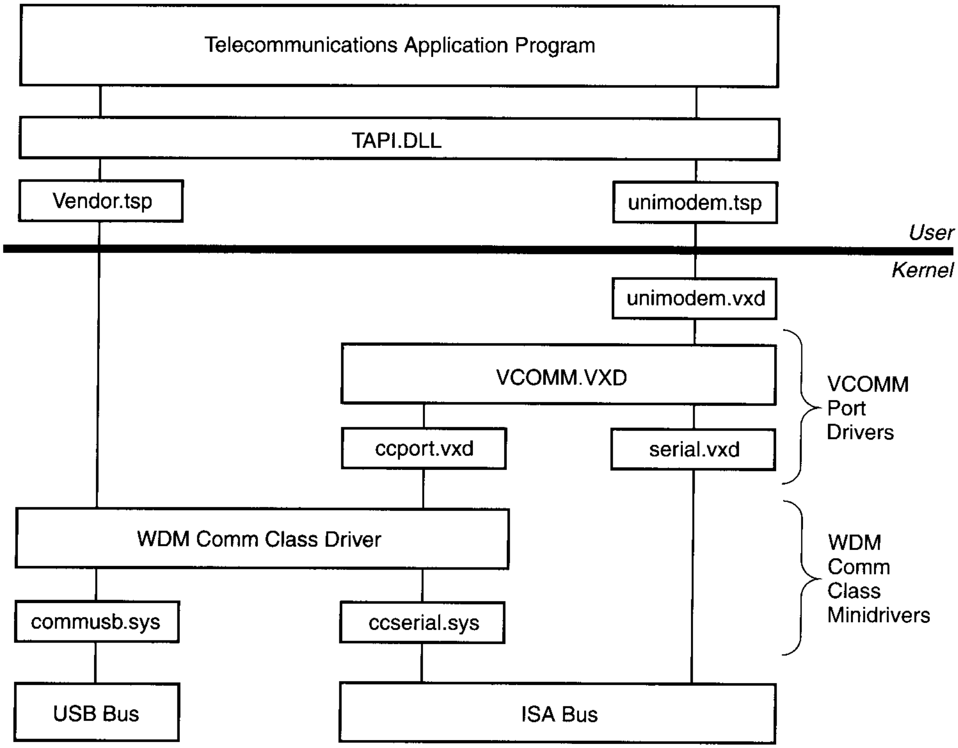


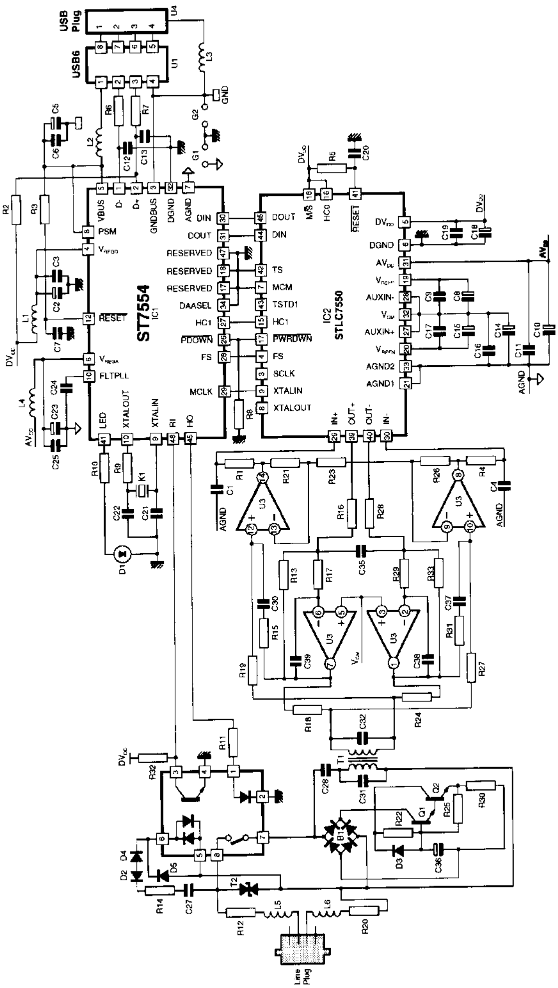
Figure 8-9. Communications stack in the Windows operating system

Removing the DSP Component

The role of the Digital Signal Processor can also be moved to the PC host processor. The modern PC host has ample performance capacity to do significant amounts of real-time signal processing besides the traditional task of supporting user applications. Intel added a set of DSP-like instructions—the MMX instructions—to the original Pentium processor; these instructions are also included in the Pentium II and Pentium III processors. The MMX instructions make short work of the DSP algorithms with the significant side benefit of being a "soft" implementation. The signal-processing algorithm is not fixed as it would be using a physical DSP component. Instead, the algorithm is loaded from disk just like every other PC host program; therefore, it can be easily changed to keep up with the latest, ever-changing communications standards. Lower cost and improved functionality with an obsolescence-proof implementation—the design is going in the right direction!

Can Anything Else be Removed or Simplified?

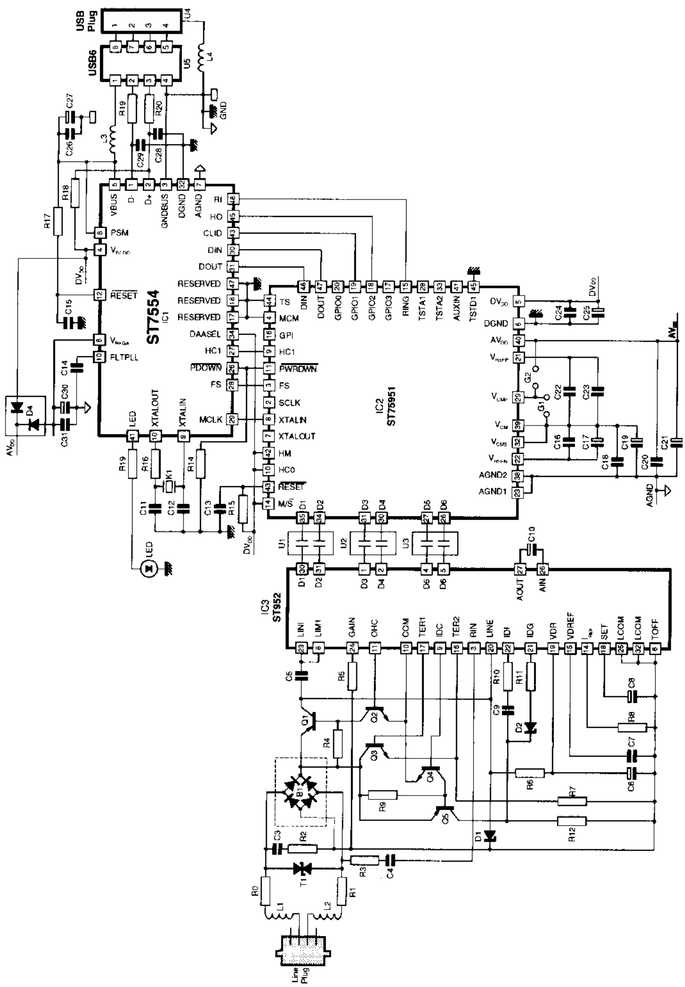
The only elements left in our modem now are the final modulation stages, signal isolation, and a data access arrangement. Figures 8-10 and 8-11 represent how this reduced design can be implemented with and without a transformer (for the full-sized schematics, see the CD-ROM). A reference design guide from STMicroelectronics details these two implementations; the guide is included in the Chapter 8/Modem directory on the CD-ROM.



Courtesy of STMicroelectronics

Figure 8-10. Modem using transformer isolation

(For the full-sized schematic, see the CD-ROM.)



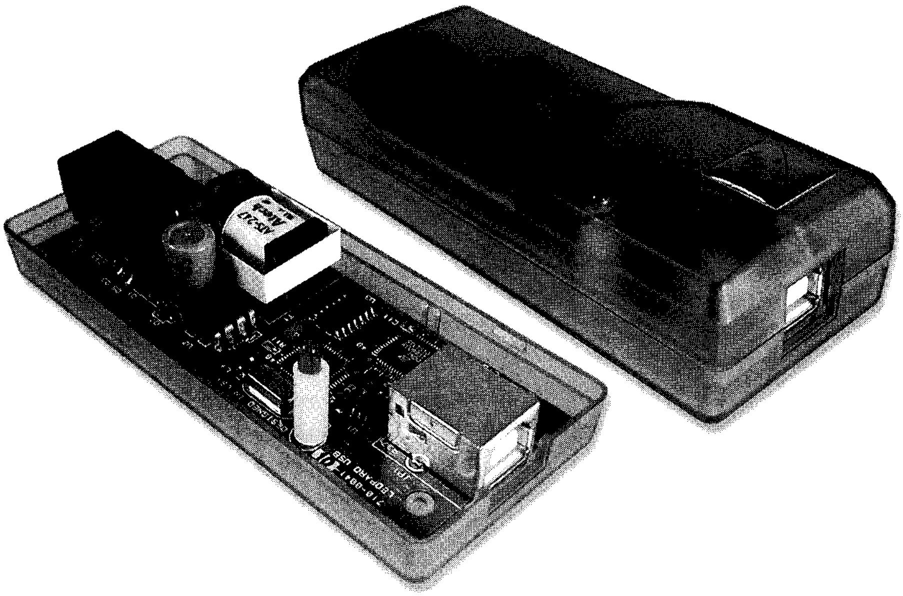
Courtesy of STMicroelectronics

Figure 8-11. Modem using a silicon data access arrangement (DAA)

(For the full-sized schematic, see the CD-ROM.)

Result of Design Optimization

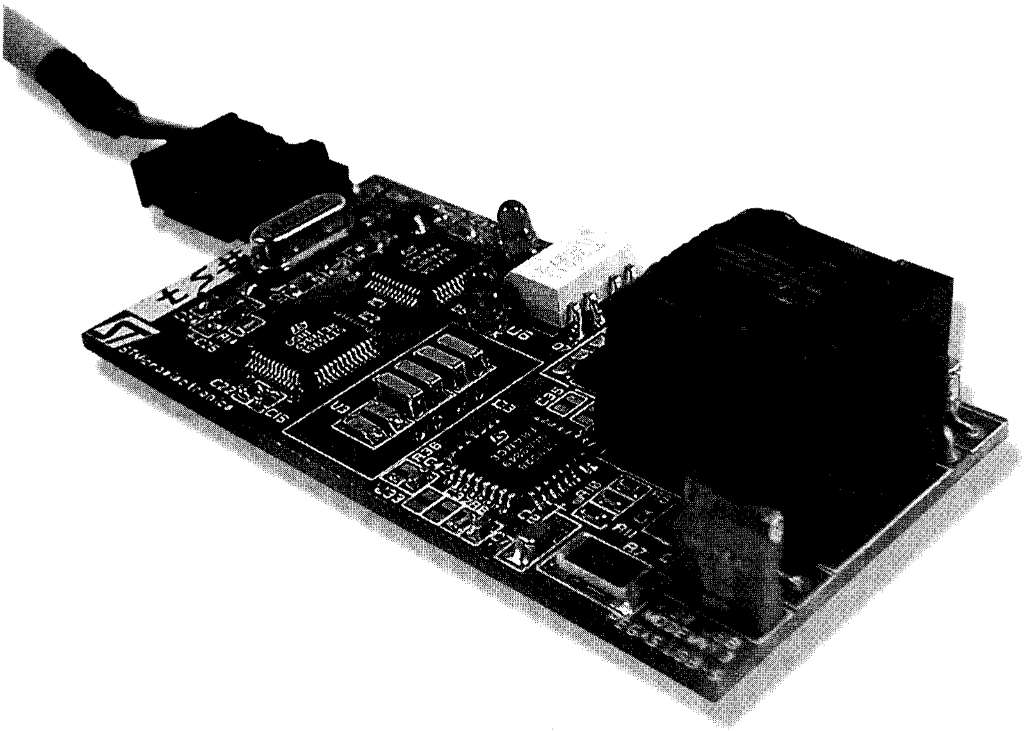
Figure 8-12 shows a very small transformer-based modem design commercially available from Shark Multimedia Inc.; this 56-Kbps fax/data modem is smaller and lighter than a box of kitchen matches.



Courtesy of Shark Multimedia, Inc.

Figure 8-12. Pocket-sized 56-Kbps modem from Shark Multimedia

Figure 8-13 shows an even smaller and lighter reference design from STMicroelectronics; the transformer has been replaced with a set of high-voltage isolation capacitors.



Courtesy of STMicroelectronics.

Figure 8-13. Transformerless reference design from STMicroelectronics

We have taken almost all the cost out of a high-performance modem by replacing most of the hardware with software on the PC host. The USB architecture facilitates this migration of hardware into software and also enables a simpler direct line model Communications Interface Class, shown in Figure 8-14.

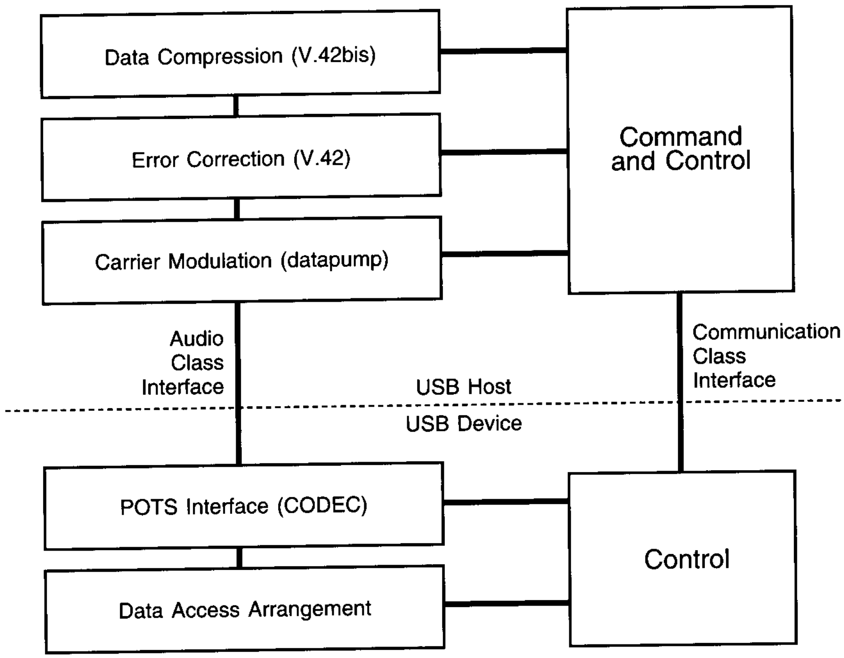


Figure 8-14. Direct line model

The Communications Interface Class uses a minimum of two pipes for control: one for the management function and the other for the notification function. The Data Interface Class also uses two or more pipes. The example in Figure 8-14 used an Audio Class Interface (see Chapter 11) to present the digitally converted data to the PC host. This type of connection would also be useful for a voice-only device such as an answering machine. A basic telephone would also require an HID interface to handle the keypad.

Table 8-2 lists the Requests and Notifications that can be made over the Communications Interface Class. As expected, the requests and notifications are telephone-line specific.

Table 8-2. Direct line model requests and notifications

Request	Code	Description	Req'd/Opt
SET_AUX_LINE_STATE	10h	Request to connect or disconnect secondary jack from POTS circuit or CODEC, depending on hook state.	Optional
SET_HOOK_STATE	11h	Select relay setting for on-hook, off-hook, and caller ID.	Required
PULSE_SETUP	12h	Initiate pulse dialing preparation.	Optional
SEND_PULSE	13h	Request number of make/break cycles to generate.	Optional
SET_PULSE_TIME	14h	Setup value for time of make and break periods when pulse dialing.	Optional
RING_AUX_JACK	15h	Request for a ring signal to be generated on secondary phone jack.	Optional
Notification			
AUX_JACK_HOOK_STATE	08h	Indicates hook state of secondary device plugged into the auxiliary phone jack.	Optional
RING_DETECT	09h	Message to notify host that ring voltage was detected on POTS interface.	Required

The Windows operating system supports all of the Communications Class devices. Today's existing telecommunications software, such as DialTone from Shark Multimedia, can immediately take advantage of these high-performance, low-cost, USB-based modems. New applications can add even more capabilities, such as voice-controlled dialing, using the extra features that the USB infrastructure provides.

Parallel Device Examples

Many parallel interfaces in a PC platform can be replaced by a USB connection. We have already discussed the legacy parallel-printer port, so this section looks at other parallel interfaces both inside and outside of a PC platform chassis.

Floppy Disk Drive

The floppy disk drive is becoming an optional peripheral device in today's PC platform. Most software is now shipped on CD-ROM, and many users are moving to Zip-style (100 MB) or Jaz-style (1 GB) removable media—more about these later. However, it is still useful to have an occasionally connected floppy disk drive that can be shared between multiple users and used as necessary.

Figure 8-15 shows the basic circuitry required to implement an attached floppy drive. The example looks just like our buttons-and-lights example but with a floppy disk controller replacing the switches and LEDs. The microcontroller firmware is much more complicated because of the inherent complexity of the floppy drive controller, but the concept of the design is the same. The I/O device identifies itself as a mass storage device with certain characteristics, and the PC host uses a mass storage device driver to send requests to the I/O device, which implements them via reading and writing from the attached floppy disk drive.

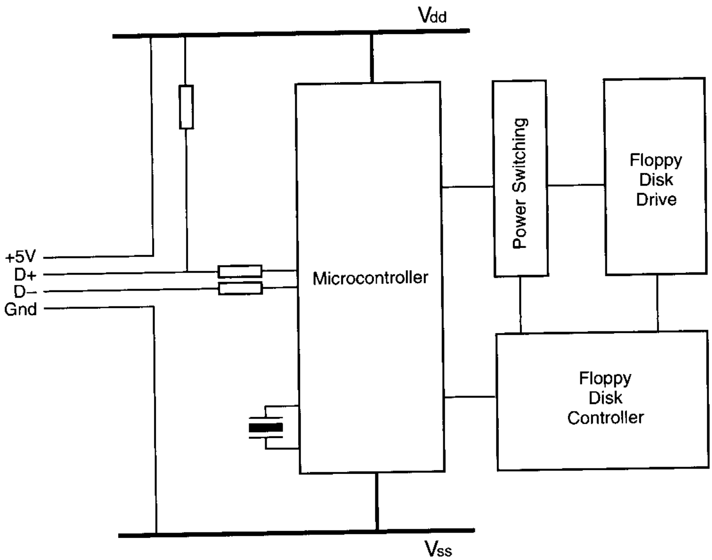


Figure 8-15. Attached floppy disk drive circuitry example

Figure 8-16 shows a commercially available product from Y-E Data. The hardware is partitioned differently, as shown in Figure 8-17, but the block diagram is the same. The unit is powered from the USB cable so no extra connections are required.



Courtesy of Y-E Data.

Figure 8-16. Floppy disk drive from Y-E Data

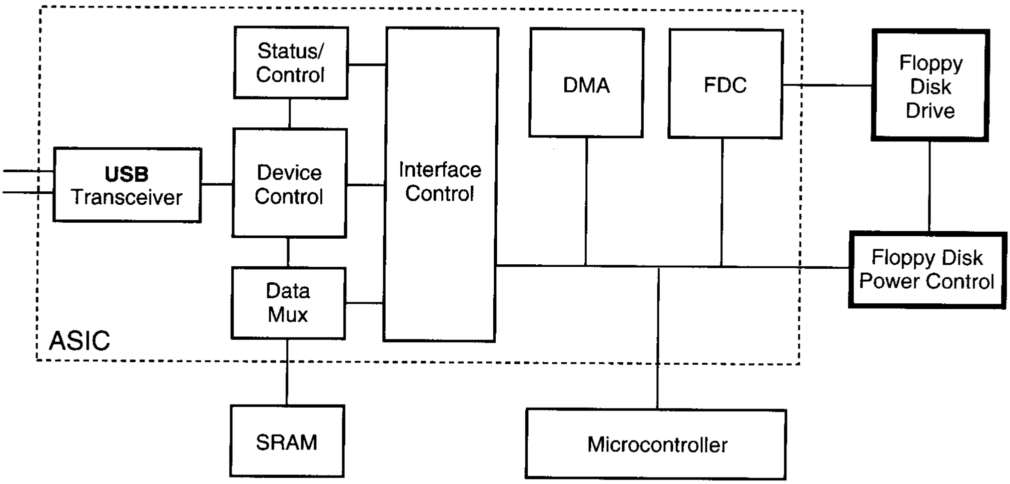


Figure 8-17. Block diagram of Y-E Data example

SCSI Devices

The SCSI parallel connection is a popular interface supporting data rates up to 5 MBps (for a standard interface, both fast and wide interfaces are available with higher data rates). A SCSI connection can be confusing because of the variety of "standard" connectors, the multitude of cables and adapters required, and the fact that signal terminators are required only at the ends of the collection of peripherals. USB is a simpler connection scheme, and manufacturers of popular SCSI devices, such as the SuperDisk drive (left) and Zip drive (right) shown in Figure 8-18, now include a USB option.



Courtesy of Imation (left) and Iomega Corp. (right).

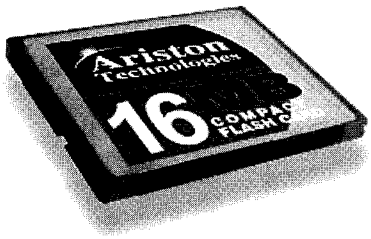
Figure 8-18. USB peripherals derived from SCSI interfaces

The SuperDisk drive uses an external bridge buried in the interconnecting cable. The smart cable is USB at one end and SCSI at the other. The SuperDisk itself is unchanged.

The Zip drive uses the embedded bridge technique via a new internal design. This new design also includes a manufacturing option of a SCSI/parallel interface.

OTHER BRIDGES

Today's megapixel digital cameras use flash memory cards to store their photographic data (Figure 8-19). Once this memory card is full, the data needs to be downloaded to a PC host for storage and/or manipulation.



Courtesy of Ariston Technologies.

Figure 8-19. Photographic data stored on flash cards

There is a lot of data to download from a camera, and a serial link running even at 115.2 Kbps can take over five minutes to download eight pictures. The delay becomes frustrating as larger memory cards store even more data. Figure 8-20 shows a better download solution that uses a USB–to–flash card converter. This interface can read and write memory cards to enable easy movement of data.



Courtesy of Ariston Technologies.

Figure 8-20. Camera memory card interface from Ariston

A block diagram of the memory card peripheral (Figure 8-21) shows a USB–to–parallel converter with the microcontroller enumerating as a bulk transfer device. Ariston supplies an application program to manage their device, and all transfers use their vendor-defined protocol because the device isn't easily defined by one of the predefined device classes.

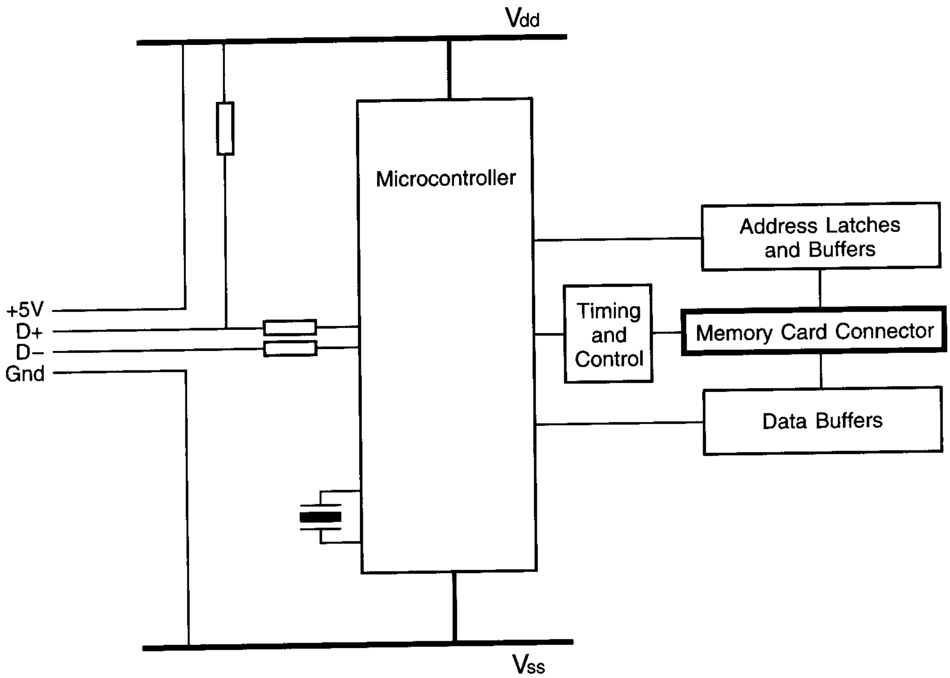
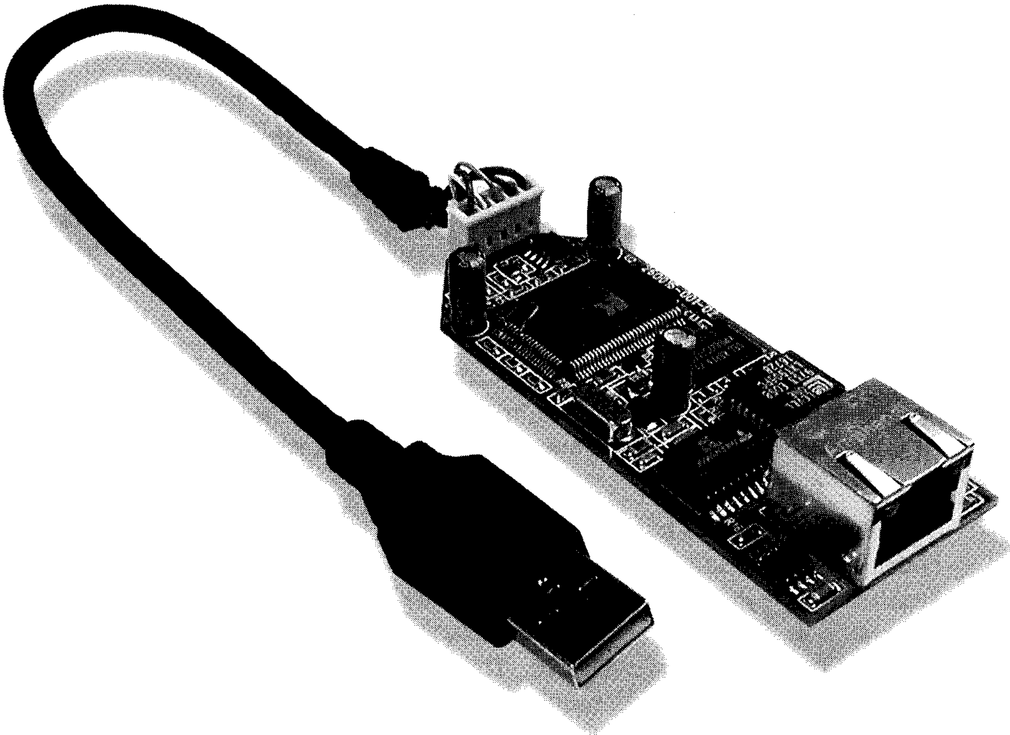


Figure 8-21. Block diagram of memory card peripheral

Another bridge component worth mentioning is the USB-to-Ethernet bridge. Figure 8-22 shows an example from ADS. This bridge has been implemented with single components to reduce costs and is not "user-programmable," as many of the other examples have been. But because its function is well defined, there would be little reason to modify these programs anyway.

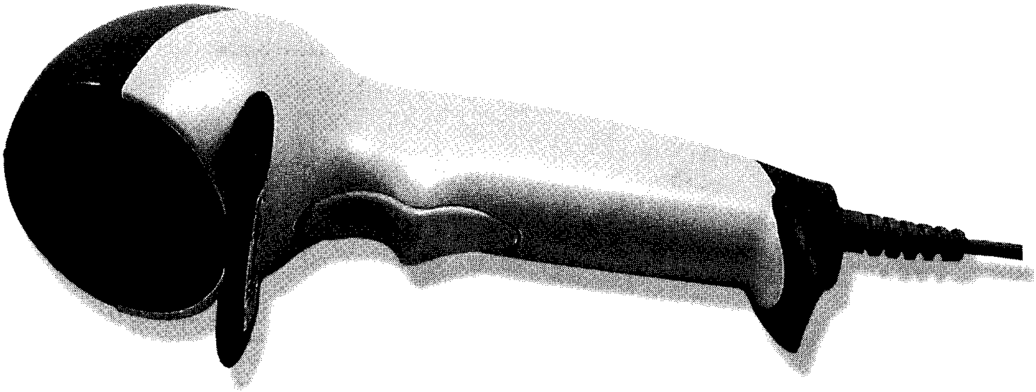


Courtesy of ADS Technologies.

Figure 8-22. USB bridges to Ethernet

BAR CODE SCANNER EXAMPLE

My final example of a USB bridge is the bar code scanner from Symbol Technologies, Inc. (Figure 8-23).



Courtesy of Symbol Technologies, Inc.

Figure 8-23. Bar code scanner from Symbol Technologies

Symbol Technologies has been a major supplier of industrial and consumer bar code scanners for many years. A Synapse proprietary protocol is used to supply scanned information to a wide variety of data processing equipment. The 6-wire Synapse connector implements a sophisticated multimaster protocol that can be interrupt-driven from both devices. The hardware interface can be minimized for volume products that do not require all of the capabilities. A software layer above this hardware provides a generic interface between a scanner and an interface cable.

Symbol decided to build an intelligent bridge between the Synapse connector and USB. The bridge uses one microcontroller to manage the Synapse protocol and another to mimic a standard USB keyboard (Figure 8-24). This design choice makes the scanner particularly easy to use in any PC environment. The scanner enumerates as a standard Windows keyboard so bar codes can be instantly read and input into any Windows application program. The USB cable provides power for the scanning motor, LEDs, and bar code scanner electronics.

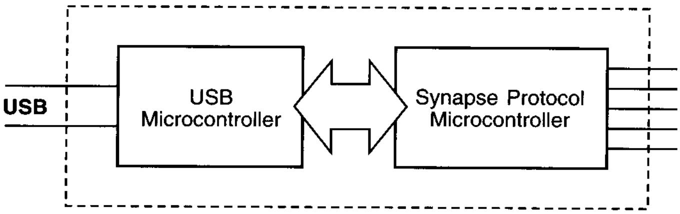
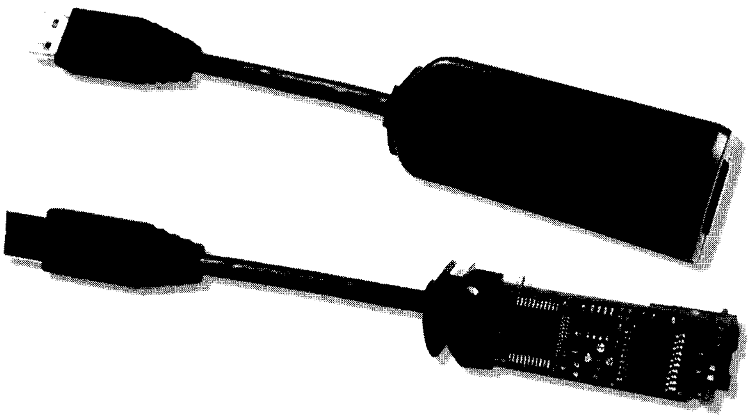


Figure 8-24. Synapse connector-to-keyboard bridge

The user is unaware of the sophistication inside the dongle. As Figure 8-25 shows, the user just attaches the dongle to the scanner and starts to use it.



Courtesy of Symbol Technologies, Inc.

Figure 8-25. Easy-to-use Synapse dongle

Chapter Summary

I hope this chapter has stimulated thoughts about how any low- to medium-speed device can be simply bridged to USB. Low speed devices use HID class drivers to minimize the amount of software that must be written. Medium speed devices should be categoried in one of the other supported classes, because this too reduces the amount of software that must be written.

This chapter described the possibility of interfacing a variety of peripheral devices to USB. Having a microcontroller in the peripheral gives us the flexibility to translate the standard reports or packets from the PC host software into custom real-world signals as required by each peripheral. This distributed intelligence approach, a key architectural feature of USB, gives a USB system its flexibility and ease of use.

Chapter 9
Connecting to the Digital World

Previous chapters have presented the bridging of USB to industry-standard computer interfaces such as RS-232, parallel, and SCSI. This chapter deals with other low-level digital interfaces such as I²C, thermometer controls, and infrared signaling.

I chose these diverse applications so I could demonstrate different techniques for implementing a variety of USB devices. Let's hope that your application is close to one of the ones covered here. All examples are implemented fully and can be found in the Chapter 9 directory on the CD-ROM.

The I²C Interface

While debugging the second example in Chapter 6, a generic microcontroller with an I²C-attached USB peripheral, I realized that I had much more than just another buttons-and-lights example. I have modified how this example is drawn (see Figure 9-1) so that the impact is clearer. Now we see an I²C bus being controlled by the PC host—to put it another way, the application program running on the PC host can send and receive I²C messages. We have an I²C development system!

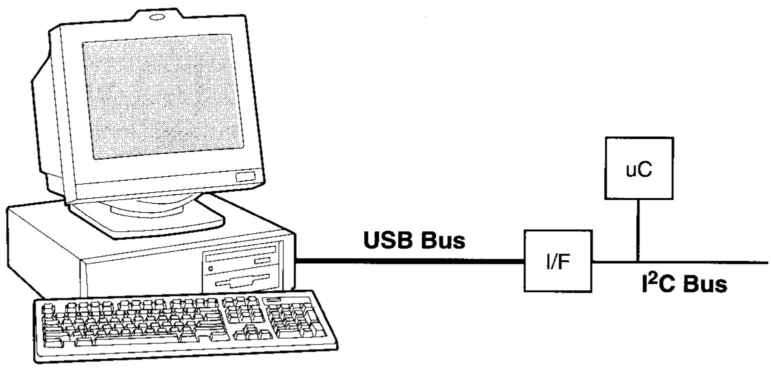


Figure 9-1. PC host as an I²C controller

The example in Chapter 6 used I^2C as a means to an end. Our goal was to control buttons and lights, and the I^2C connection was a convenient method to implement a solution. If we shift our focus to the I^2C bus, we can change the PC host application program to be a potent I^2C development system.

Before getting too involved in the details, I'll provide an overview of the I^2C bus architecture. You will discover many diverse components that use an I^2C interface, so the ability to control them from a PC program will be a very useful tool.

The I^2C Specification

The I^2C bus was developed by Philips Semiconductors over 20 years ago as a serial Inter-Integrated Circuit bus. The bus uses two wires (Figure 9-2) to implement a simple, bidirectional, multimaster bus. The full specification is included in the Chapter 9/ I2C directory on the CD-ROM; I have provided highlights from the specification below.

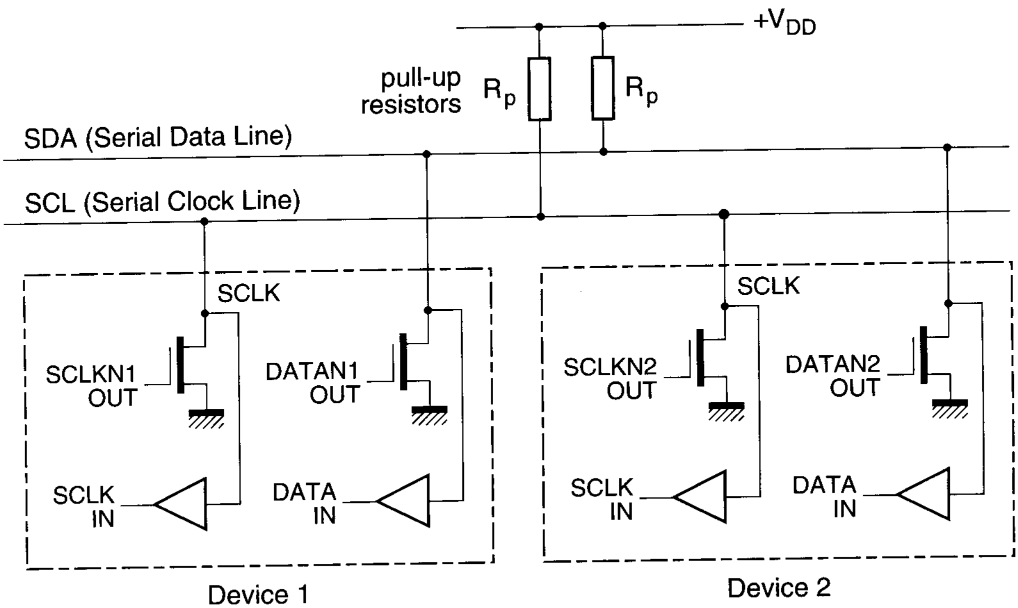


Figure 9-2. Connection of I^2C components to I^2C bus

The I²C bus is a "wired-OR" bus. A resistor pulls up the Serial Clock Line (SCL) and the Serial Data Line (SDL), and any device connected to the bus can pull the signal low. A rigid protocol is built on top of this simple signaling scheme to implement efficient intercomponent communication. The I²C bus is a multimaster bus, but only one master can ever be in control. The current master can pass control to an alternate master using a defined handoff sequence.

The master is responsible for generating the clock signal. Data bits are transmitted in 9-bit frames that are marked by a START and STOP condition (Figure 9-3). The master is responsible for generating the start and stop conditions. Data is allowed to change on the bus while the clock signal is low; data is considered valid when the clock signal is high. A slow I/O device can hold the clock low to give itself more time to operate—this is called "clock stretching."

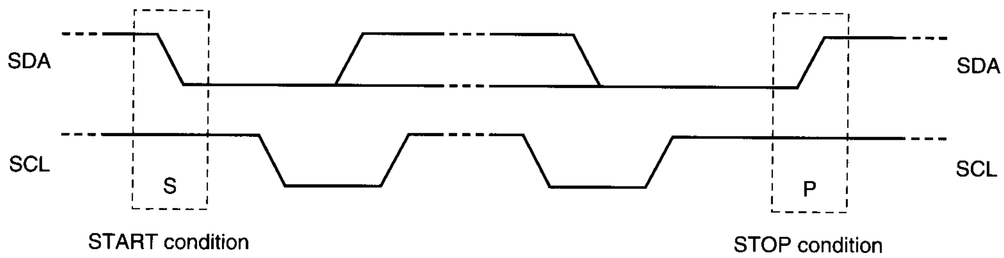


Figure 9-3. Bit-level signaling on I²C bus

The data transfer is as follows:

1. A master always outputs a 7-bit address and a read/write bit in its first frame. All I²C devices have preassigned family addresses; for multiple identical components, such as memory, local jumpers set the low part of the address. Thus, each device on an I²C bus will have a unique address.
2. The master waits for an acknowledge from the addressed slave before proceeding.
3. The listener acknowledges the receipt of the byte and, by driving the data line low, gives the master permission to send the next byte.
4. Data is transferred in the next frame from the master to the slave for a write and from the slave to the master for a read.
5. The master can initiate another data transfer or signal a STOP condition.

More data transfers use subsequent addresses. In each case, the data receiver acknowledges the receipt of the data by driving the data line low. Figure 9-4 shows this acknowledge sequence.

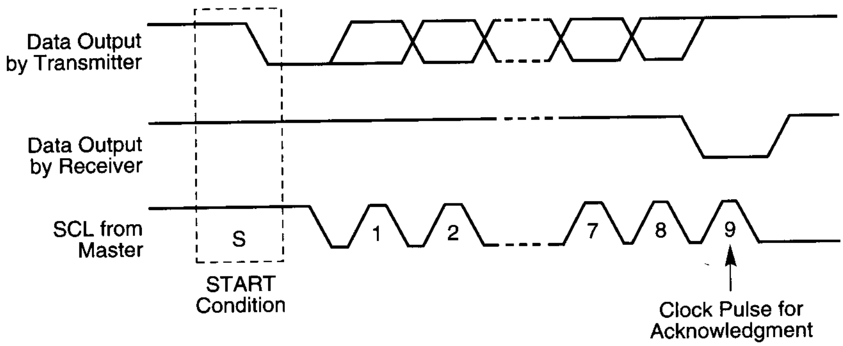


Figure 9-4. Data handshake via acknowledge

Some microcontrollers have an integrated I2C buffer and sequencer and can operate at the standard 100-kHz rate or the faster 400-kHz rate. Microcontrollers without this buffer and sequencer can generate the clock and data sequences with only a small amount of programming. For more detail, see the Application Note in the Chapter 9/I2C directory on the CD-ROM.

Figure 9-5 shows a simple I2C controller application. The application is written in Visual Basic, and the source is included in the Chapter 9/I2C directory on the CD-ROM. You can initiate I2C sequences from the PC host and see responses displayed on the screen. Many I2C devices require extensive initialization, so I added a capability to read long sequences from a text file. Long response sequences can also be saved to a text file.

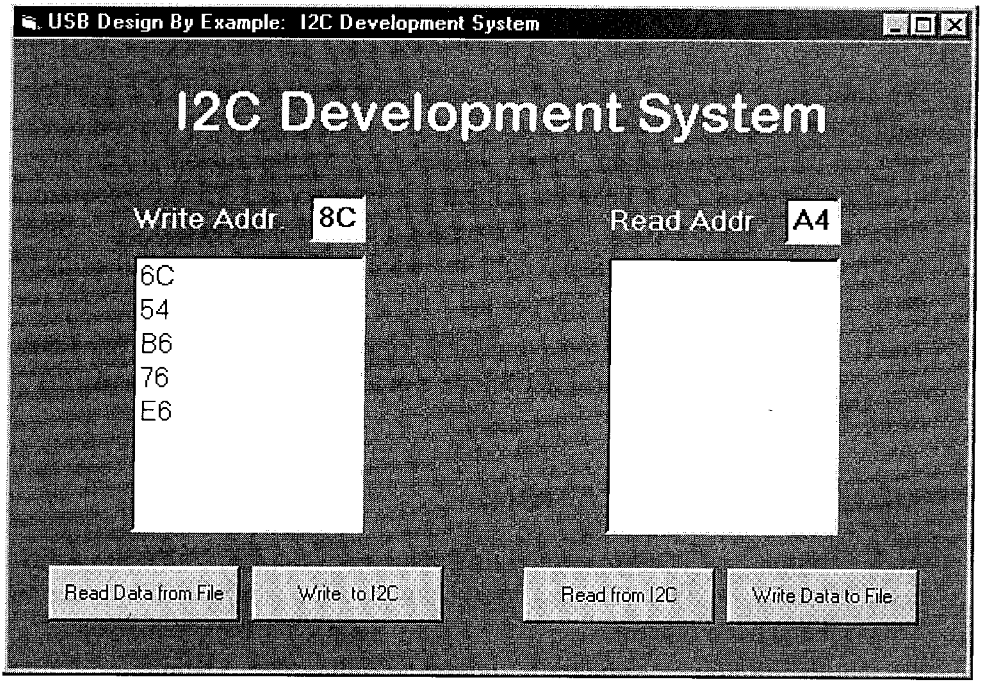


Figure 9-5. I²C development application

The range of I²C devices that we can connect to the microcontroller is extensive. A typical piece of consumer equipment, like a television or a VCR, is made up of building blocks interconnected with an I²C bus. Look on www.philips.com for a wide selection of parts that you could consider using with this USB-based example.

For more information about the I²C bus, I recommend *The I²C Bus, from Theory to Practice*, by Dominique Paret, published by Wiley with ISBN 0-471-96268-6.

I²C Summary

We built an USB-to-I²C converter using a single component. In the example, I used the Anchor Chips USB microcontroller; its integrated I²C support made the solution particularly elegant. Other USB microcontrollers also have an I²C interface; those that don't can implement one in software. The microcontroller firmware was almost trivial, and its execution used almost no processor time—which means that an I²C bridge could be added to almost every USB example to give excellent added value.

The I²C interface is used extensively in the consumer industry, and our USB-to-I²C dongle allows the PC host to become a potent I²C prototyping and development system.

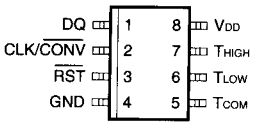
THERMOMETER APPLICATIONS

In this section, we'll design several thermometer examples. Each uses a different serial protocol, but this will be masked from the PC host application program. This is the general process:

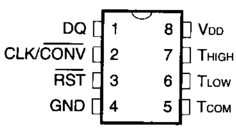
1. A uniform byte-oriented buffer is exchanged between a Windows application program and the USB microcontroller.
2. The application program reads and writes temperatures and limits as simple byte variables.
3. The microcontroller implements the bit handling and protocol required to actually read a temperature or write a limit.
4. The uniform byte-oriented interface will essentially decouple the hardware solution from the software solution so that each can evolve separately. I'll demonstrate this hardware independence by changing only the hardware in the second example.

Example 1: Reading Temperatures

Figure 9-6 shows our first thermometer example using the Dallas Semiconductor DS1620. The product operates using a 3-wire interface and a control register. For the full data sheet, see the Chapter 9/Thermometer directory on the CD-ROM.



DS1620S 8-Pin SOIC (208 MIL)



DS1620 8-Pin PDIP (300 MIL)

Pin Description

DQ	-	3-wire Input/Output
CLK/CONV	-	3-wire Clock Input and Stand-alone Convert Input
RST	-	3-wire Reset Input
GND	-	Ground
THIGH	-	High Temperature Trigger
TLOW	-	Low Temperature Trigger
TCOM	-	High/Low Combination Trigger
VDD	-	Power Supply Voltage (3V-5V)

Figure 9-6. DS1620 digital thermometer

The operation of the 3-wire interface is similar in principle to the I2C interface:

1. A CLK is generated by a master.
2. Data is exchanged on a wired-OR DQ line. DQ data is valid for read and write transactions on the rising edge of CLK.
3. A third signal, RST, must be high for the DS1620 to be enabled.
4. Once triggered, a DS1620 takes approximately 400 ms to create a valid result (worst case is 1000 ms), so we'll poll at 100-ms intervals.

The components are interconnected with the CLK and DQ lines bused and separate RST lines (Figure 9-7). The RST signal is essentially a chip-select.

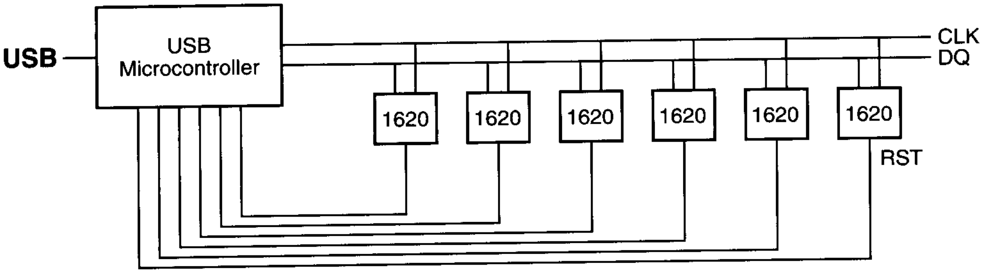


Figure 9-7. Using six thermometers

Fixed format packets are sent to a DS1620 to initiate an action. This example sets up the six thermometers operating in parallel in continuous temperature conversion mode. The USB microcontroller poll eachs thermometer and store the value locally.

The PC host application is written in Visual Basic, and its human interface is shown in Figure 9-8. The full source code is provided on the CD-ROM.

The interface to the PC host is an 8-byte buffer that can send all of the temperatures to the host on request. If the temperature values have not changed since the previous poll by the PC host, then the request is NAK'ed to preserve USB bus bandwidth.

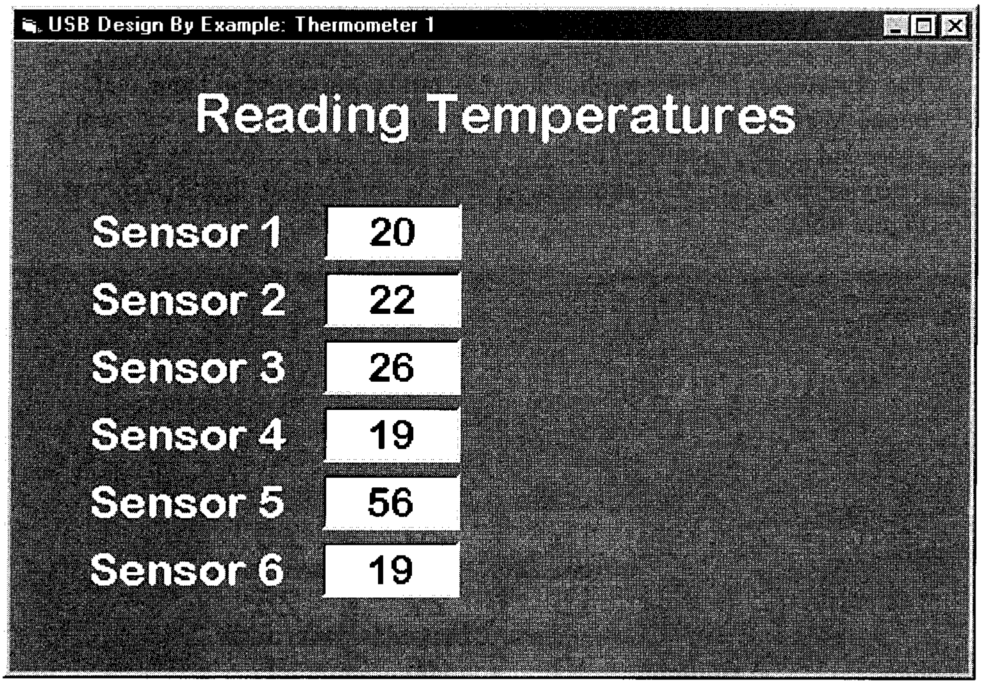


Figure 9-8. Human interface for the temperature sensor

Most of the firmware is identical to the buttons-and-lights example in Chapter 5. The report format is different, and the TIMER routine that enters data into the report is different. We'll change our product name in the Device Descriptor and make some minor changes to the initialization, and we're done! The full source code for this firmware is included on the CD-ROM. The thermometers are initialized to operate in continuous conversion mode, and we will poll them every 100 ms. Each thermometer will update its internal copy of temperature asynchronously to our polling.

Example 2: Adding Temperature Limits

The controlling PC host application may not be interested in the actual temperatures at each sensor but would be most concerned if any of them exceeded some predefined limits. The human interface from the first thermometer example is extended to include temperature limits as shown in Figure 9-9.

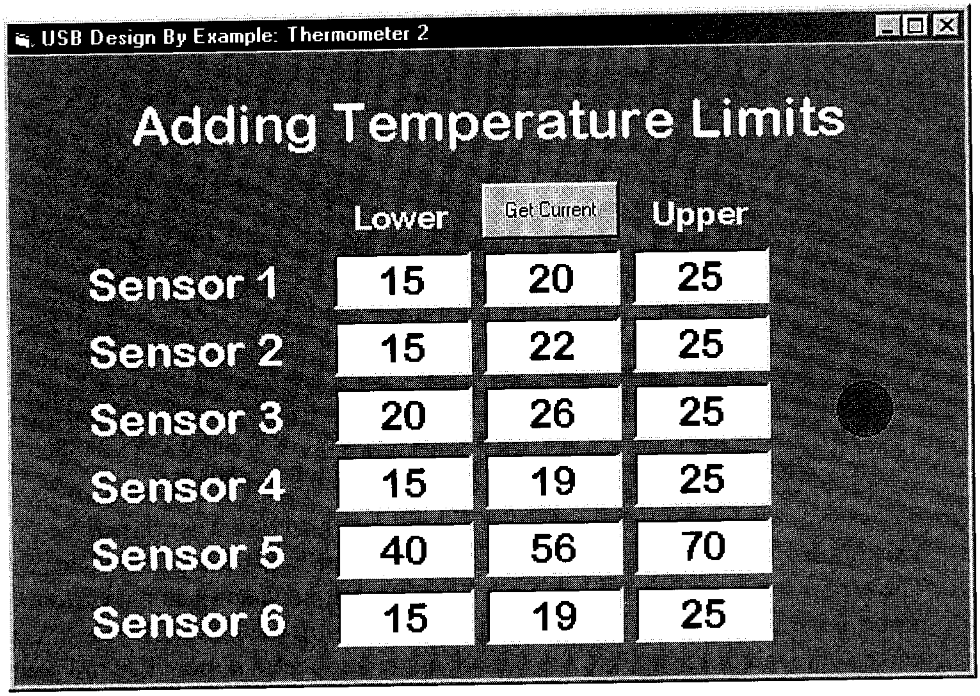


Figure 9-9. Adding temperature limits to PC host application

Although the DS1620 can detect out-of-range results locally, the following example provides out-of-range detection using firmware in the USB microcontroller.

In this second example, I have defined two more reports:

- Output report: used by the PC host application to update the temperature limits on each thermometer.
- Alarm input report: alerts the PC host application that a problem is developing. The USB microcontroller checks the temperature limits during each TIMER polling routine and enables the transmission of the alarm input report if required.

The source code for the PC host application program and the microcontroller firmware can be found in the Chapter 9/Thermometer/Example 2 directory on the CD-ROM.

Example 3: Using a Multidrop Thermometer

Our first two examples required multiple wires between the USB microcontroller and the thermometer. Each DS1620 requires a 3-wire signaling interface and two more wires for power and ground. This wiring can be cumbersome for more than six to eight thermometers. So, let's look at a multidrop thermometer, called the DS1820, also from Dallas Semiconductor (Figure 9-10).

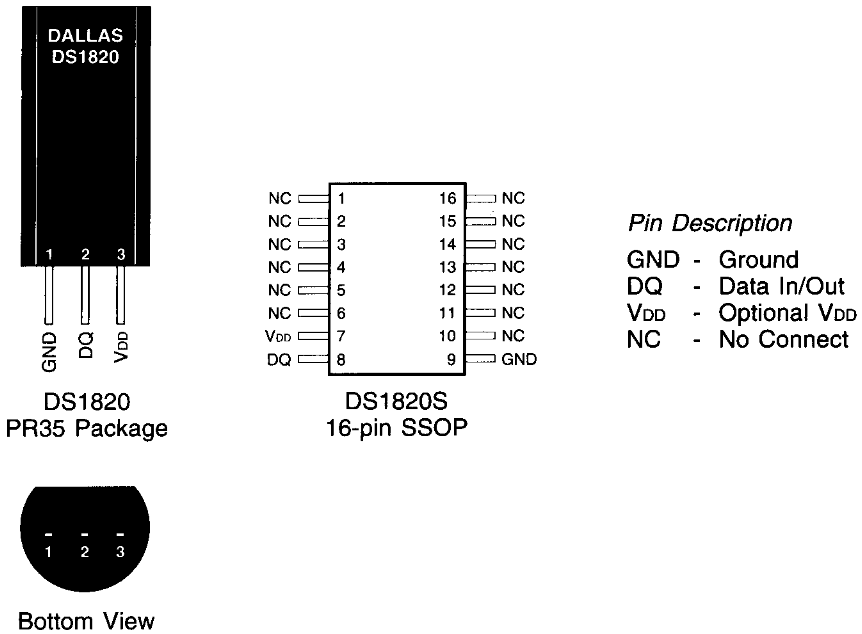


Figure 9-10. DS1820 multidrop thermometer

The DS1820 is a member of the MicroLAN family, which requires that only two wires be bused between all devices. One wire is GROUND, and the other wire is sometimes POWER and sometimes SIGNAL. Each MicroLAN device collects power from the POWER line and saves it in a local capacitor. When a device needs to signal, the master temporarily removes the power source so that signaling can take place and then restores power once signaling is complete. This complicates the protocol somewhat, but the savings and simplicity in wiring are worth the complexity. For a full MicroLAN tutorial, see Chapter 9/MicroLAN directory on the CD-ROM.

Each MicroLAN device is manufactured with a unique 64-bit identifier used for communication with devices on a MicroLAN. The technology has applications in much wider fields than our third thermometer example, and the USB-to-MicroLAN controller we are about to develop can be used in this much broader context.

The hardware for our third example, as shown in Figure 9-11, is simpler than the previous example.

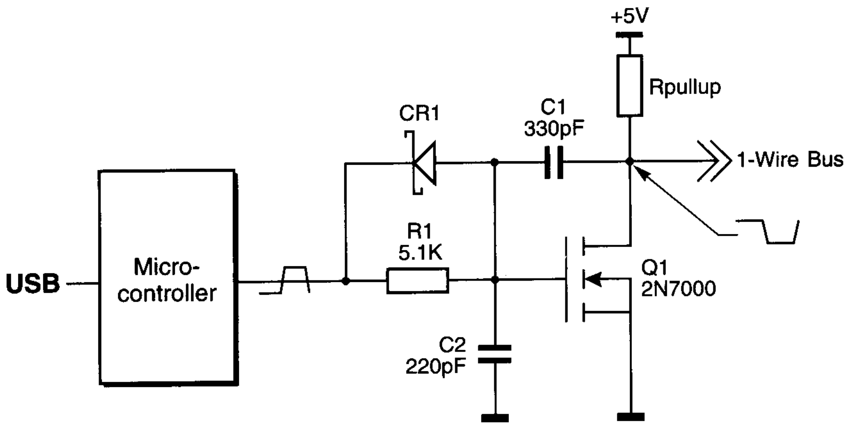


Figure 9-11. USB-to-MicroWire converter

The initialization firmware is more complicated because the ID of each of the DS1802 thermometers must be discovered, but the polling routine in TIMER is basically the same. The report structures do not need to change, so the PC host application does not need to change either—yet another example of swapping out the "real-world" electronics (in this case, replacing the DC1620s with DS1820s) and not having to change the PC host software at all.

Thermometer Applications Summary

The thermometer examples show how the USB microcontroller could implement an arbitrary serial protocol and mask changes in this protocol by providing a simple byte-oriented interface to the PC host applications software. The low-level bit-handling details were off-loaded from the PC host (which is very poor at this task) and implemented efficiently by the attached USB microcontroller. Furthermore, temperature limit checking was also off-loaded to the microcontroller, and the PC host was interrupted only when a policy decision was needed.

USB makes it easy to add intelligent slaves that will share an overall task, and this means better use of the PC host. See the next chapter for a larger example.

INFRARED SUBSYSTEMS

Laptop computers have had infrared signal generators and detectors for some time. The desktop PC, however, has missed out on the versatility of this signaling scheme. In this section we'll design an USB-to-infrared dongle that supports the PC-industry IR standards and the consumer industry IR remote controller implementations.

PC-industry IR Standards

Infrared has been used by the PC industry to solve two different needs. The first was a need to transfer large amounts of data from a PC to a peripheral such as a printer. This file-oriented data transfer has also proved useful for PCs and personal data assistants (PDAs; for example, the Palm Pilot) to exchange files. The standard is called IrDA Data, and the full specification, provided by the Infrared Data Association (www.irda.org), is included in the Chapter 9 directory on the CD-ROM.

The second need involved data transfer to and from wireless peripherals such as keyboards, mice, and game pads. The requirements of this data transfer are quite different from those of IrDA Data, so a different specification, called IrDA Control, was developed for this application. This specification is also included on the CD-ROM.

Unfortunately, the same piece of hardware cannot be used for both standards. Even the LEDs are different because of different data transfer requirements. However, we'll discover that the hardware requirements are quite modest and easily implementable on a USB dongle. The hardware for IrDA Data can also generate and receive consumer IR (i.e., a TV remote control), and this will be described in a later section.

Example 1: IrDA Data

Infrared communication relies on line-of-sight data transmission between a transmitter and a receiver. In a typical desktop environment, this line-of-sight could be easily broken (by a coffee cup in the way, for example), so the IrDA group also defines a set of communications protocols above the physical layer to ensure reliable transfer of data.

- The IrDA 1.0 specification has an upper speed limit of 115.2 Kbps, because this is the fastest that an asynchronous UART can operate.
- The IrDA 1.1 specification describes operation up to 4 Mbps while maintaining compatibility with Version 1.0 products. A synchronous UART is required for this.

Both solutions will be presented. The IrDA protocol stack (Figure 9-12) is fully implemented in Windows 98 and other operating systems. It is straightforward to redirect the physical layer to a USB dongle.

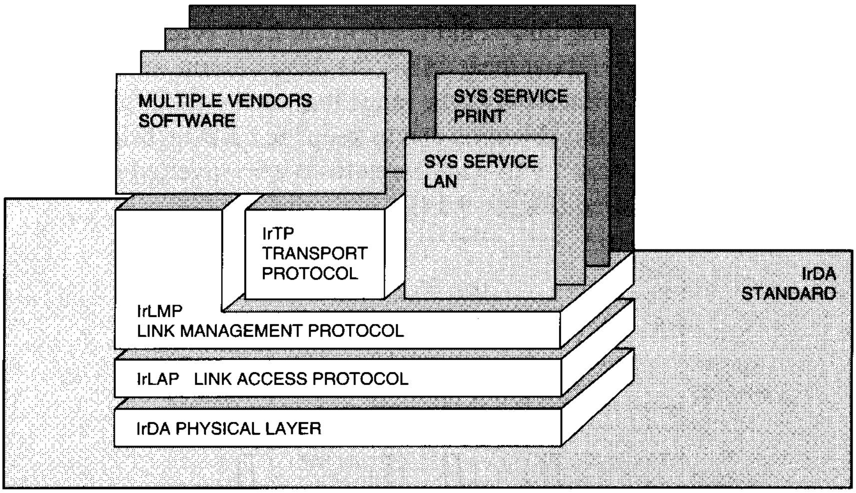


Figure 9-12. IrDA protocol stack

The IrDA Data 1.0 physical layer was designed as a low-cost add-on to existing PC hardware. Figure 9-13 shows a typical implementation on a PC serial port.

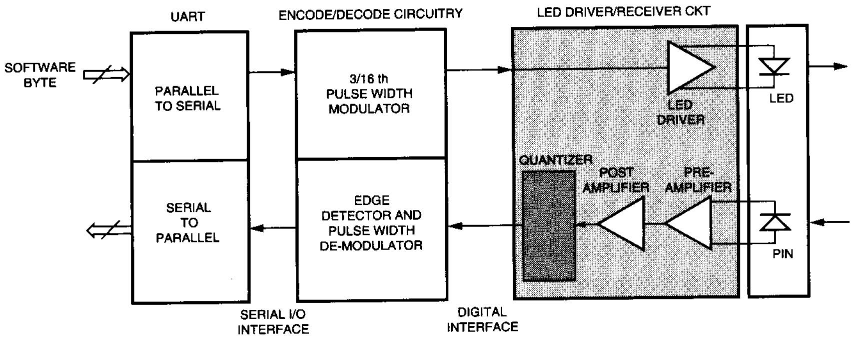


Figure 9-13. Implementation of IrDA Data 1.0 physical layer

The serial bit stream from the UART is not used directly to turn on the transmitter LED, because this would result in the LED being on for long periods of time. The LED would overheat unless the drive current were reduced. But reducing the drive current reduces the range that can be used for successful detection. To resolve this dilemma and to keep the LED on brightly for a shorter amount of time, a narrow (3/16 width), return-to-zero inverted encoding and decoding scheme is used (Figure 9-14).

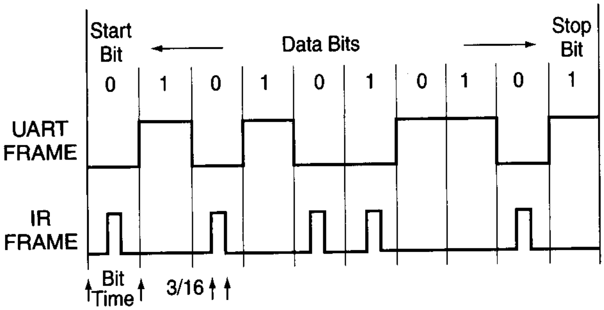


Figure 9-14. Encoding scheme used for IrDA 1.0

The encoder/decoder (sometimes called an "endec") could be an external component, such as Hewlett Packard's HSDL-7000, or it could be integrated into the UART component. An external encoder/decoder requires a 16x clock that is used to correctly position the LED activation signal with a bit time.

The IrDA 1.0 specification has an upper speed limit of 115.2 Kbps, because this is the fastest that an asynchronous UART can operate. The IrDA 1.1 specification describes operation up to 4 Mbps, and a synchronous UART is required for this. A different LED encoding scheme, Pulse Position Modulation (PPM), is also used because it has twice the data density. Rather than encoding a single bit in a 3/16 pulse within a bit time, PPM encodes two bits within a 500-ns bit time (Figure 9-15).

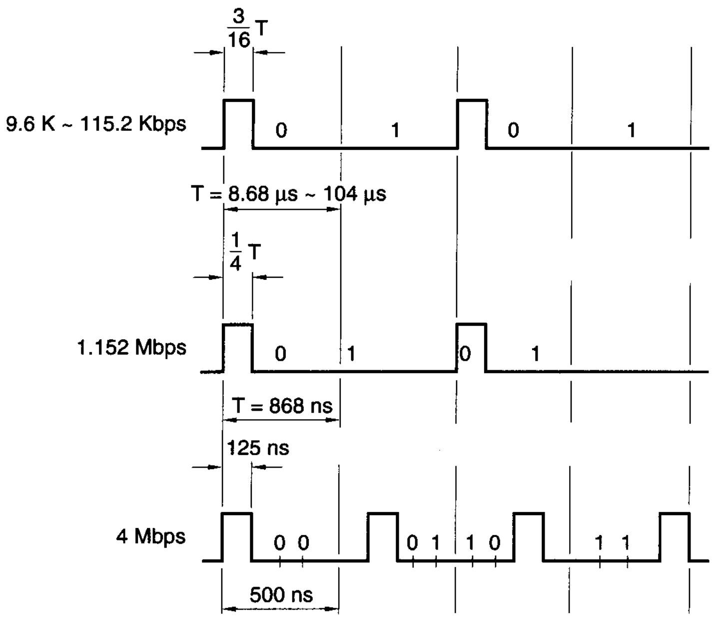


Figure 9-15. Encoding scheme used for IrDA 1.1

An IrDA 1.1 encoder/decoder is not available as an external component; it is only available integrated into a Synchronous Serial Communications controller (SSC). There are two options to obtain an IrDA 1.1–compliant SSC controller:

- Several manufacturers produce IR controllers (IBM, SMSC, and TI; for data sheets, see the Chapter 9 directory on the CD-ROM).
- A larger number of manufacturers produce Super-I/O components that include an IrDA 1.1–compliant SSC controller. One might think that the Super-I/O component, which also includes a floppy disk controller, keyboard, mouse, and serial and parallel ports, would cost more, but because these products are produced in much higher volumes for the PC market segment, this solution is actually cheaper.

A 4-Mbps link requires a byte of data every two microseconds. Typically this is too fast for a microcontroller to move bytes under program control, so some DMA scheme is required to maintain this throughput. Figure 9-16 shows an example of the hardware using an SMSC USB evaluation board. The board uses a 97C100 microcontroller and an FDC37C67X super I/O controller.

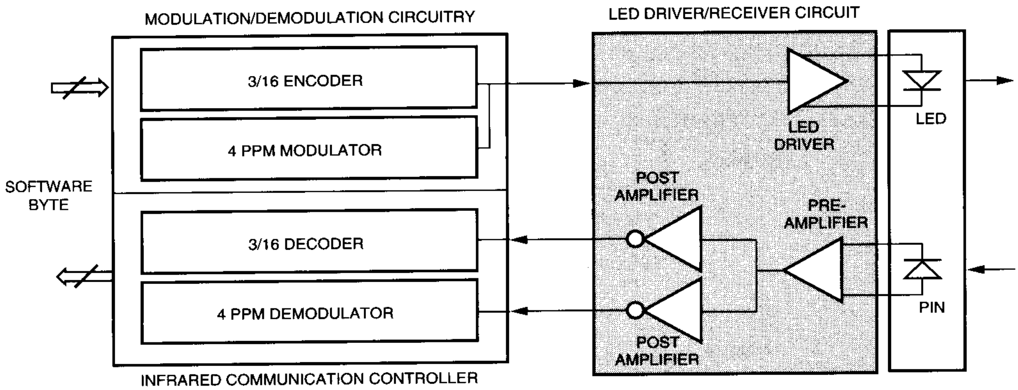


Figure 9-16. Fast IR physical layer

Example 2: IrDA Control

IrDA Control is quite different from IrDA Data technology. While IrDA Data is a peer-to-peer, file-oriented data transmission system, IrDA Control is a master-slave, command-and-control transmission system. IrDA Control is specifically designed to pass short control packets between a host and a collection of wireless I/O devices. The standard is not designed to move large amounts of data.

A PC host will poll up to eight wireless devices in sequence to determine if they have any control information to provide. Up to four of these eight devices can be active at any time. IrDA Control devices have a range of up to seven meters, so a mouse, keyboard, or remote control can interact with a PC at the other end of the room.

An IrDA Control system is defined in layers as shown in Figure 9-17. IrDA Control devices are a good match to USB HID devices, and in this section we will build an IrDA-Control dongle.

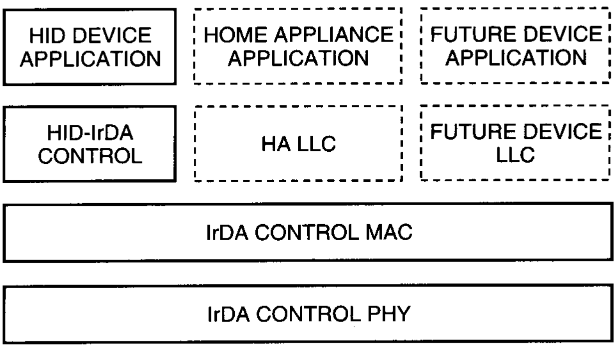


Figure 9-17. IrDA Control layered implementation

The IrDA Control Specification (included on the CD-ROM) defines the transmission schemes, modulation schemes, and wavelengths of the infrared transmitter and receiver. A serial data stream uses a 16 Pulse Sequence Modulation (16 PSM) format to modulate a 1.5-MHz carrier frequency to achieve an overall data throughput of 75 Kbps.

A 16 PSM encoder takes four information bits at a time and creates a Data Symbol that it transmits over the optical link (Figure 9-18). Similarly, the decoder re-creates a serial bit stream, four bits at a time, from received Data Symbols.

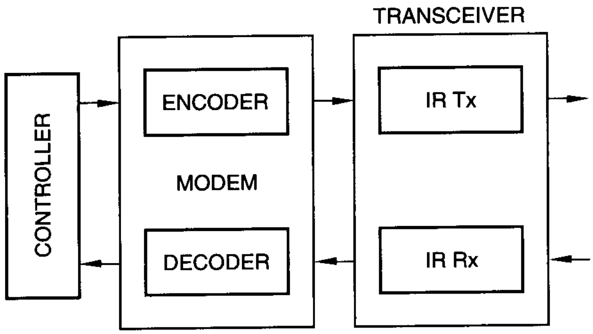


Figure 9-18. IrDA Control physical layer block diagram

The serial bit stream is generated by the Media Access Layer in frames (Figure 9-19). Following a preamble to synchronize the receiver host address, peripheral address and control information are sent with the payload data. This is similar to the USB interrupt packet technique.

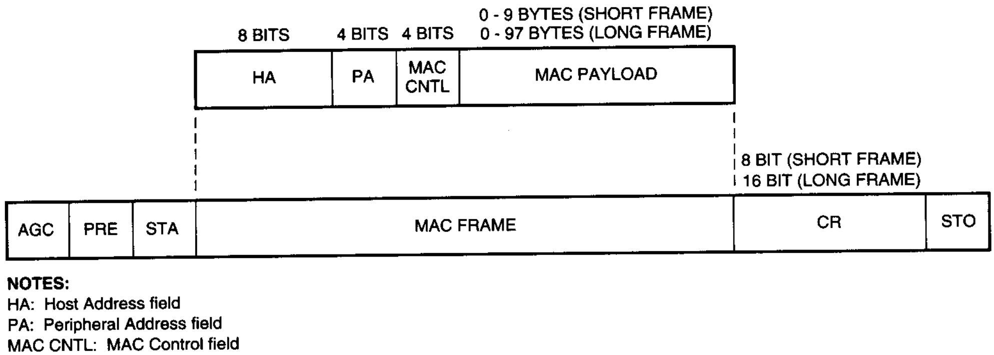


Figure 9-19. IrDA Control MAC frame structure

Because the IrDA Control specification is relatively new, there is a limited choice of hardware to implement a reference design. The single example I found is very complete, however. The Sharp LZ85202 is a full USB–to–IrDA Control subsystem needing only a USB transceiver (for example, PDIUSBPD11 from Philips) and an infrared transceiver (for example, CP2W2001YK from Sharp) to implement the complete hardware design (Figure 9-20). Figure 9-20 also shows the hardware/firmware required at the wireless peripheral.

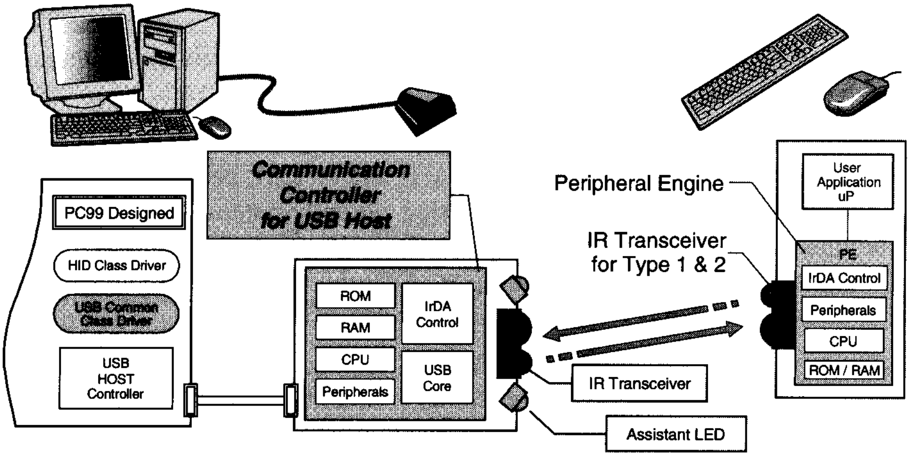


Figure 9-20. Complete USB–to–IrDA Control subsystem

Example 3: Consumer Industry IR

It is difficult to find standards for IR remotes from the consumer industry. All vendors seem to do their own designs, and the infrared systems from different manufacturers are rarely interoperable. A growing industry is now providing "universal" or "learning" remotes so that a single, complex unit can take the place of several simple units. This section looks inside a "universal" remote, and we'll design an IR receiver/transmitter that allows a PC to respond to an IR remote and also generate IR remote signals to control consumer electronics components. With a PC host added to the control loop, we can implement more sophisticated and user-friendly sequences.

Almost all IR remotes transmit bursts of modulated infrared light. The light is modulated at a frequency between 30 kHz and 60 kHz, because this reduces the interference effects from ambient lighting. A few remotes, I am told, do not modulate their outputs and just send pulses of light—this is supposed to save battery power and help reduce the cost of the device. All the remotes I tested used modulated light, and no two used the same frequency.

The bursts of modulated light and no light encode serial data. The length of the burst, or the spacing between the bursts, or both, is different for a logical 0 and a logical 1. Figure 9-21 shows a representative signal transmitted from a remote when a key is pressed.

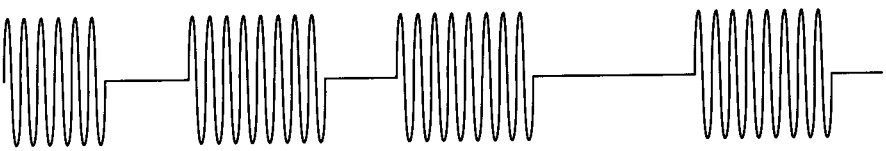


Figure 9-21. Consumer IR signal sequence

Our controller will operate in two distinct modes or configurations. The first mode will MEASURE the signals from an existing IR remote controller and will create a Translation Table. This table is then provided to the OPERATE mode, which sends and receives IR signals.

The measure mode uses several tasks to successfully decode the signal from a given remote controller to reproduce its signal:

- Determine modulation frequency
- Characterize the pulse and gap lengths
- Determine the Command Length

We have a lot of counting to do! A microcontroller could easily control a known IR remote in software. Detecting an unknown high frequency is much more difficult for the microcontroller because of the granularity of instruction timings. To successfully detect frequencies in the range of 30 to 60 kHz, we need a hardware timer and some kind of capture mechanism (Figure 9-22).

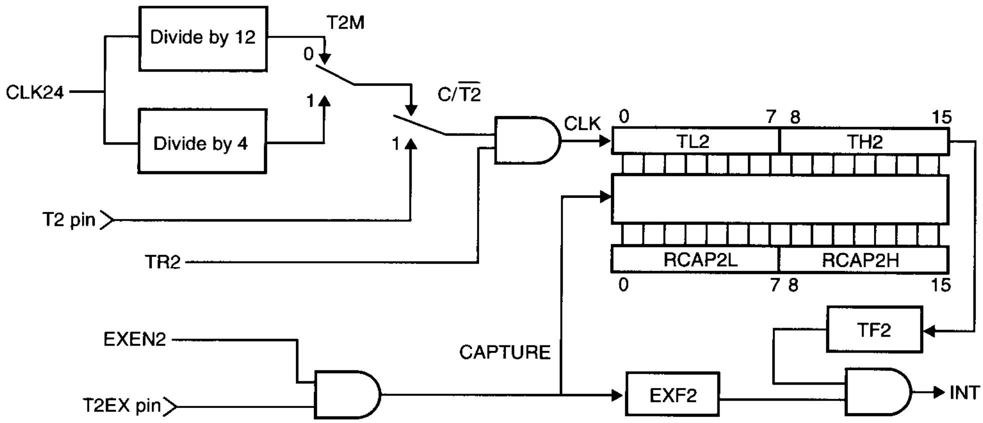


Figure 9-22. Using a timer to measure pulse width

Most microcontrollers have an integrated timer, which can be used for this pulse-width type of measurement. The Anchor Chips EZ-USB has two timer/capture registers, but we use only one. The timer needs to be clocked at a rate that will give us good discrimination between adjacent frequencies. Here is the overall process:

1. The value of the timer is saved.
2. On the transition of the input signal, the microcontroller is interrupted and saves this value.
3. At the next transition of the input signal, the timer is captured and the microcontroller is interrupted again.
4. The microcontroller subtracts the stored value from the new value to determine the width of the input pulse.

In our example we will measure all of the input pulses that represent frequencies between 30 kHz and 60 kHz and average them to best determine the modulation frequency. This frequency range corresponds to 33.3 ms to 16.6 ms, and we should use a timer frequency that produces about 100 sample points between these two extremes. The example program collects pulse width data in a histogram format and stops collecting if an overflow is detected on one of the 8-bit counters, or if the input signal expires.

If a pulse width outside of this 33.3 ms to 16.6 ms range is measured, then we have measured the interburst gap. Subtracting saved values will also produce the width of the burst. This detail is shown in Figure 9-23.

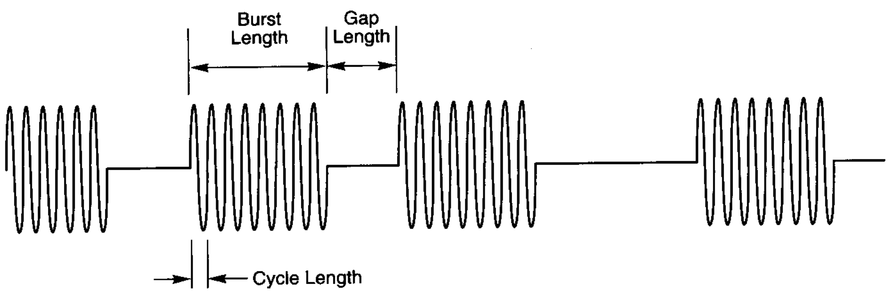


Figure 9-23. Three key measurements are made

The example program saves the burst length and gap length in a table and then calculates the next burst and gap lengths. Measurements are made until the input signal expires or our table is full. The collected information is supplied to the PC host in a measurement report.

The PC host application program analyzes this data to determine the modulation frequency and the 1 and 0 encoding scheme. A translation table is built in which the names engraved on the IR remote buttons are paired with a hexadecimal representation of the transmitted signal stream, for example, "Volume Up = 03F4."

Once all of the IR remote buttons have been decoded, the translation table is downloaded to the microcontroller during reconfiguration.

The microcontroller is now switched out of MEASURE mode into OPERATE mode by choosing a different CONFIGURATION. A single entry report can now be exchanged between the PC host and the microcontroller. IR signals detected by the microcontroller are decoded and sent to the PC host as "button press detected"; the PC host will send "button requests" to the microcontroller, which will generate the correct IR signal sequence.

The example on the CD-ROM also includes the feature of saving and reading translation tables from disk so that the MEASURE mode need be used only once per IR remote type.

Infrared Subsystems Summary

We learned that there are three basic types of infrared that a PC host can comprehend. IrDA Data is used for short-range peer-to-peer file transfer, and we built a USB dongle to demonstrate this. IrDA Control is used to manage a collection of slave peripherals in the same room as a PC host, and we built a USB dongle to demonstrate this. There is a wide variety of infrared controls (e.g., TV remote) available with no industry standards. We built an adaptive infrared receiver that could analyze and then re-create the signals for any selected remote. We built a USB dongle to demonstrate this.

In all cases the USB microcontroller managed the low-level encoding and decoding of the infrared signal while the PC host did the data processing. We did not meet the goal of having a single USB dongle to implement all of the infrared solutions, but the examples presented can be integrated, in a building-block fashion, into a variety of solutions.

CHAPTER SUMMARY

The common thread throughout this chapter has been the USB microcontroller's ability to generate an arbitrary serial waveform using any protocol, yet present a byte-oriented interface to the PC host. This partitioning of the task between the data processing abilities of the PC host and real-time responses of the USB microcontroller is an area in which the intelligent slave architecture of USB can be very effective. USB makes this style of system implementation both easy and hardware-independent. The software and hardware can evolve and improve at their own rates, and, providing the software interfaces remain constant, the USB solution will continue to operate.

CHAPTER 10 CONNECTING TO THE REAL WORLD

The world of digital electronics operates at very low power levels. A typical microcontroller output is capable of sinking 5 mA from a 5-volt source—this is only 25 mW. While this is enough to trigger another electronic part and just enough to light a small LED, it is not enough to control the real world. Some kind of amplification is required, and this will be the first topic of this chapter.

A typical microcontroller input can discriminate between a logical 0 and a logical 1. Logical 0 is between 0 and 0.7 volts at less than 1 mA, and a logical 1 is between 2.4 and 5.0 volts at less than 1 mA. Signals in the real world are not like this; they vary from very small millivolt signals up to hundreds of volts on heavy equipment. These signals are often continuously variable and do not have convenient logical 0 and logical 1 values. Some kind of signal conditioning must be done before the microcontroller can accept this signal, and this will be the second major topic of this chapter.

OUTPUT SIGNAL CONDITIONING

A transistor can be used to amplify the power of a microcontroller output signal as shown in Figure 10-1.

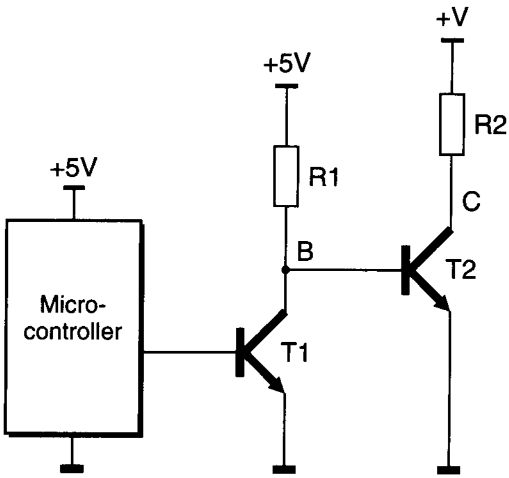


Figure 10-1. Switching larger power levels

When the microcontroller output is LOW, that is, close to 0 volts, then transistor 1 is OFF. Current is supplied by R1, which turns transistor 2 ON. Assuming transistor gains of 100, transistor 2 could be switching 10 amps through its load resistor R2.

When the microcontroller output is HIGH, that is, close to 5 volts, then transistor 1 is turned ON. The voltage at point B will drop to 0.1 volts, which will in turn cause transistor 2 to turn OFF. The 10 amps that was flowing through R2 now stops.

The simple transistor circuit shown in Figure 10-1 allows the power switching of a microcontroller output to be amplified by 1000 times. R2 does not need to be connected to the same +5-volt supply as the microcontroller—it could be connected to a higher DC voltage supply that further increases the power levels that can be switched.

The load resistor R2 does not need to be a resistor as shown in Figure 10-2—it could be an indicator lamp or a relay that could switch even higher power levels.

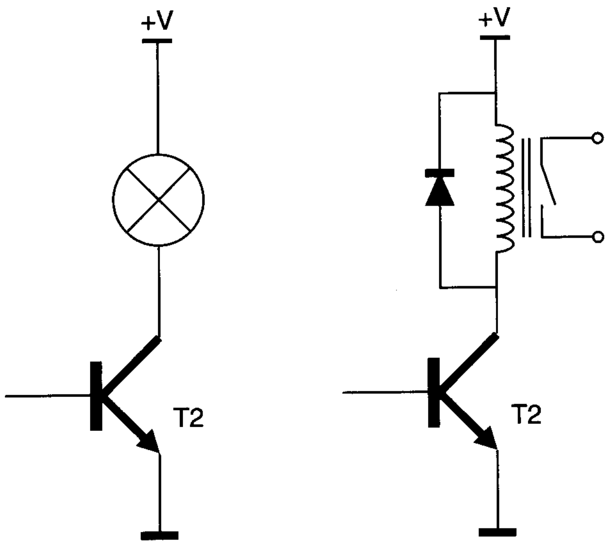


Figure 10-2. Switching high power loads

The diode across the coils of the relay of Figure 10-2 prevents high reverse voltage from occurring across transistor T2 when the transistor turns off. The inductance of the relay coils will resist the current flow from being turned off and will cause a high reverse voltage, called the back electromotive force (EMF), to be generated at point C. The diode safely conducts the high-current pulse away.

Another benefit of the relay circuit is the isolation of the load and its supply voltage from the microcontroller circuitry. Potentially harmful (to the microcontroller) voltages are prevented from coming into contact with the control circuitry.

Driving coils is not limited to relays; an electric motor is also made up of coils. We have already learned enough to control large coil currents, so let's look at what a small amount of logic can do for motor control.

CONTROLLING A MOTOR

Electric motors, whatever their size, come in two basic types: stepper motors that rotate in discrete steps and DC motors that rotate at a variable speed. First we'll control a stepper motor, and then we'll move on to a DC motor.

Example 1: Stepper Motor

Figure 10-3 shows a simplified diagram of a stepper motor. There are two sets of motor windings, called Phase A and Phase B, and two poles. Real stepper motors have many sets of poles, which results in less rotation for each "step": The simplified two-pole stepper motor will rotate 360/4 = 90 degrees for each full step while a typical twelve-pole stepper motor will rotate 360/24 = 15 degrees for each full step. The following discussion uses a two-pole example, so remember to divide the step size by the number of real poles on your stepper motor.

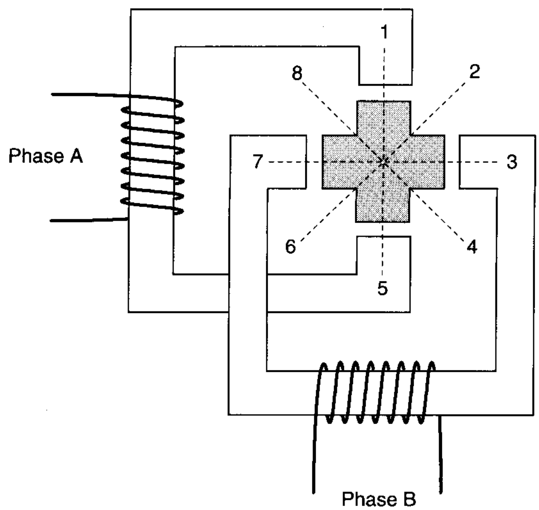
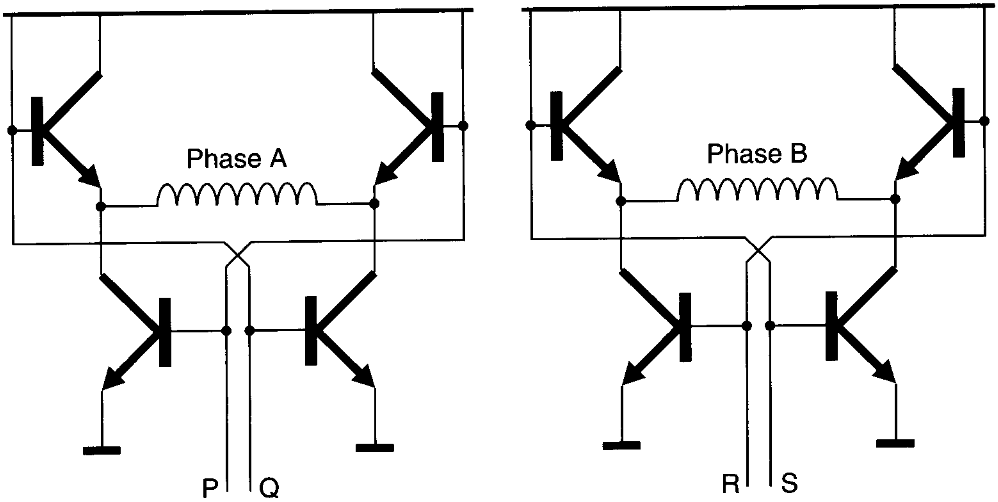


Figure 10-3. Simplified stepper motor

The rotor of the stepper motor is stable in many discrete orientations as indicated by the numbered positions in Figure 10-3. To move the rotor to an adjacent stable position, Phase A and Phase B are activated in a sequence.

In Figure 10-4, part A shows the connection of power transistors used to drive the motor coils. In the simplest case, P or Q ON and R or S ON, current flows through each coil left-to-right or right-to-left. This gives four possible states and results in four possible positions for the rotor (Figure 10-4, part B).



A. Power transistors drive the coils



P	ON	OFF	OFF	ON
Q	OFF	ON	ON	OFF
R	ON	ON	OFF	OFF
S	OFF	OFF	ON	ON

B. Position of rotor is determined by control signals

Figure 10-4. Moving rotor to adjacent stable position

Moving from one stable position to the next stable position is called a "step." It is possible to create another mode for each of the coils by defining more states where P and Q are OFF or R and S are OFF. This results in stable half-step positions as shown in Figure 10-5.



P	ON	OFF	OFF	OFF	OFF	OFF	ON	ON
Q	OFF	OFF	ON	ON	ON	OFF	OFF	OFF
R	ON	ON	ON	OFF	OFF	OFF	OFF	OFF
S	OFF	OFF	OFF	OFF	ON	ON	ON	OFF

Figure 10-5. More control states produce a half-step

A problem with this half-step solution is uneven torque created by one or two coils ON at any instant—the full-step example always has two coils on. The uneven torque can create vibration and/or mechanical noise. The solution is to reduce the current drive when both coils are ON or overdrive the current through a single coil when it is ON. This will produce a constant torque and will be more reliable.

If we are prepared to make our control circuitry more complicated (hey, we have a microcontroller, so this is NO PROBLEM), we can define more stable states. This technique, called microstepping, involves varying the current drive into each coil by discrete amounts. So rather than ON + OFF, which is 100% + 0%, we could define

- 100% + 66% + 33% + 0% for quarter steps, or
- 100% + 92.4% + 83.1% + 70.7% + 55.5% + 38.2% + 19.5% + 0% for eighth steps

Figure 10-6 shows an example of an Allegro 2916 controller, which supports quarter steps. The microcontroller moves up the sequence for clockwise rotation and down the sequence for counterclockwise rotation.

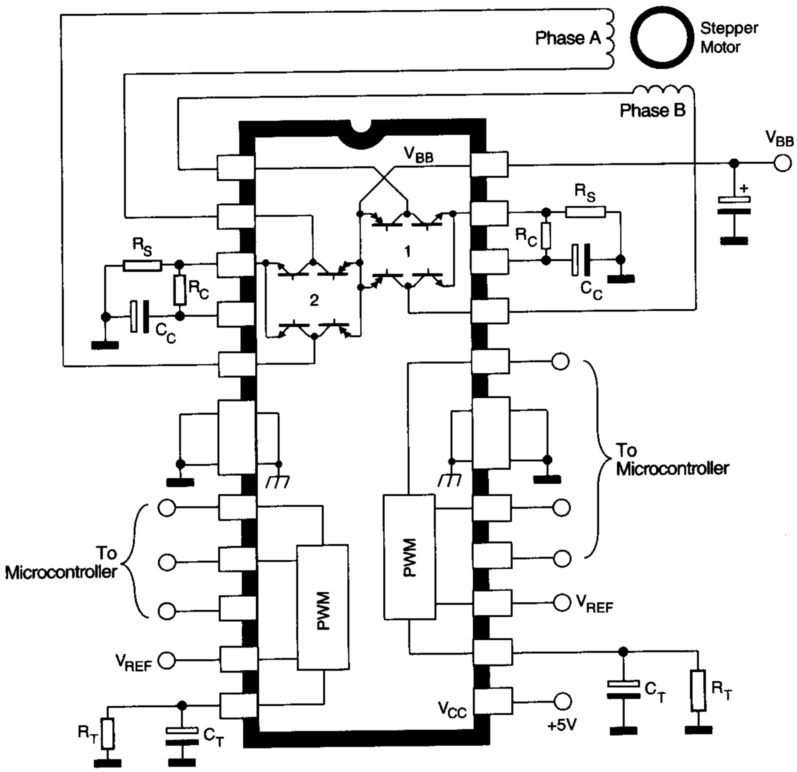


Figure 10-6. Stepper motor controller from Allegro

A full working example is included in the Chapter 10 directory on the CD-ROM. The beauty and simplicity of a stepper motor control system is that it is reliable enough to operate "open-loop." Open loop means that the microcontroller does not need to sense where the rotor is—the stepper motor reliably and repeatedly moves to the next stable position following a sequence command. Worm gears or toothed belts are often used to translate the stepper motor's rotational accuracy to linear positioning accuracy.

Example 2: DC Motor

Some applications do not require precise position control, but they do require precise speed control. This can be achieved with a cheaper DC motor. The speed of a DC motor is proportional to the voltage applied across its terminals; this assumes that the voltage is within the designed operating range of the motor. The direction of rotation of the motor can be changed by swapping the input drive signals. High-power transistors are required to drive the coils of a DC motor, and the circuit shown in Figure 10-4 for a stepper motor will also operate a DC motor. Because there is only one coil on a DC motor, only one set of power transistors is used.

The torque of a DC motor is also a function of its input voltage, so if we require reliable starting even at low speeds, we should use the maximum voltage allowed for the motor. If this maximum voltage were maintained, then the motor would rotate at its maximum speed. If a slower speed is required, the supply voltage is removed and then periodically reapplied again as shown in Figure 10-7.

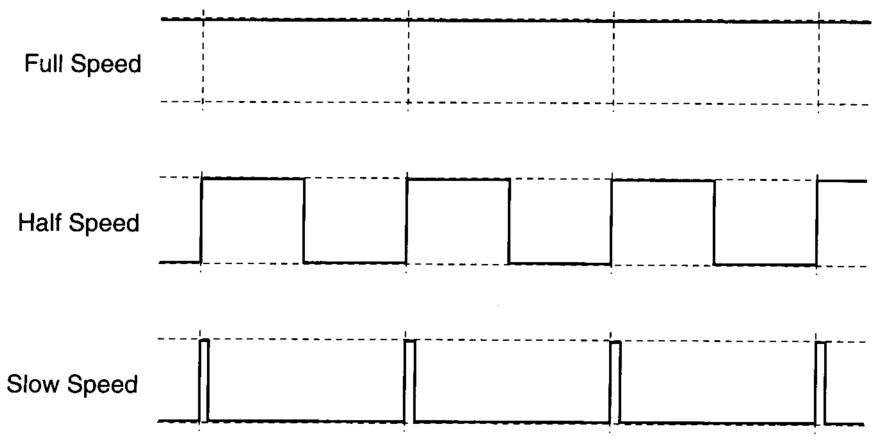


Figure 10-7. DC motor speed control

The technique shown in Figure 10-7 is called pulse width modulation, and the speed of the motor is determined by the **average** voltage level. It is difficult to predict the final speed for a given input voltage, so "closed-loop" feedback is often used with a DC motor system. In a closed-loop system, some kind of sensor would be included on the shaft of the motor that provides feedback to the microcontroller so the microcontroller can determine the exact speed of rotation. Noncontact, Hall-effect devices are often used for this application; several data sheets are included in the Chapter 11 directory on the CD-ROM.

CONTROLLING A LINE-POWERED DEVICE

While a relay coil can be used to control a LINE-powered device, it is not the preferred method. A transistor-based solution will provide superior performance, be more reliable, and be price competitive.

Example 3: Lighting Panel

First let's address the relay's major attribute of signal isolation. Figure 10-8 shows an alternate method of implementing signal isolation using an optoisolator. The load seen by the microcontroller is a current-limited light-emitting diode that can be turned on by a 5-mA output signal. The light causes the transistor to turn on, and this in turn can switch higher power loads.

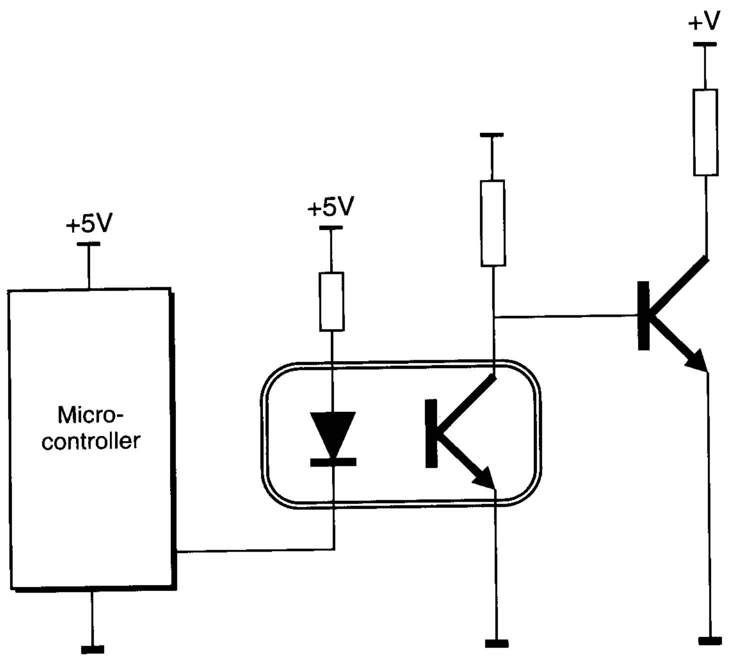


Figure 10-8. Isolating control and load signals

Particularly important when the microcontroller is switching high-power loads such as AC LINE signals, signal isolation adds an important safety aspect. The alternating voltage also gives us more opportunity to control the output power—the output load is switched on and off every power cycle, and the ratio of the ON/OFF time gives us 0 to 100 percent continuously variable power control with a single ON/OFF signal (Figure 10-9). Note that the OFF time occurs at the next zero crossing.

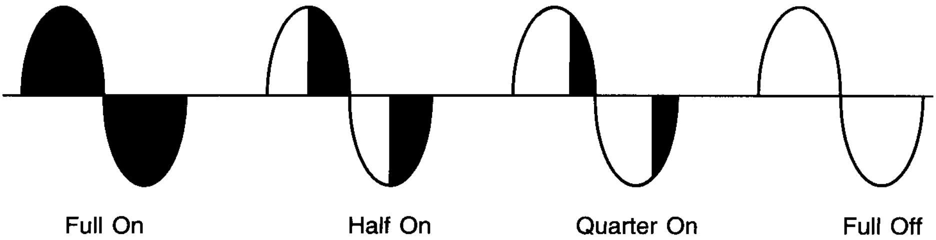
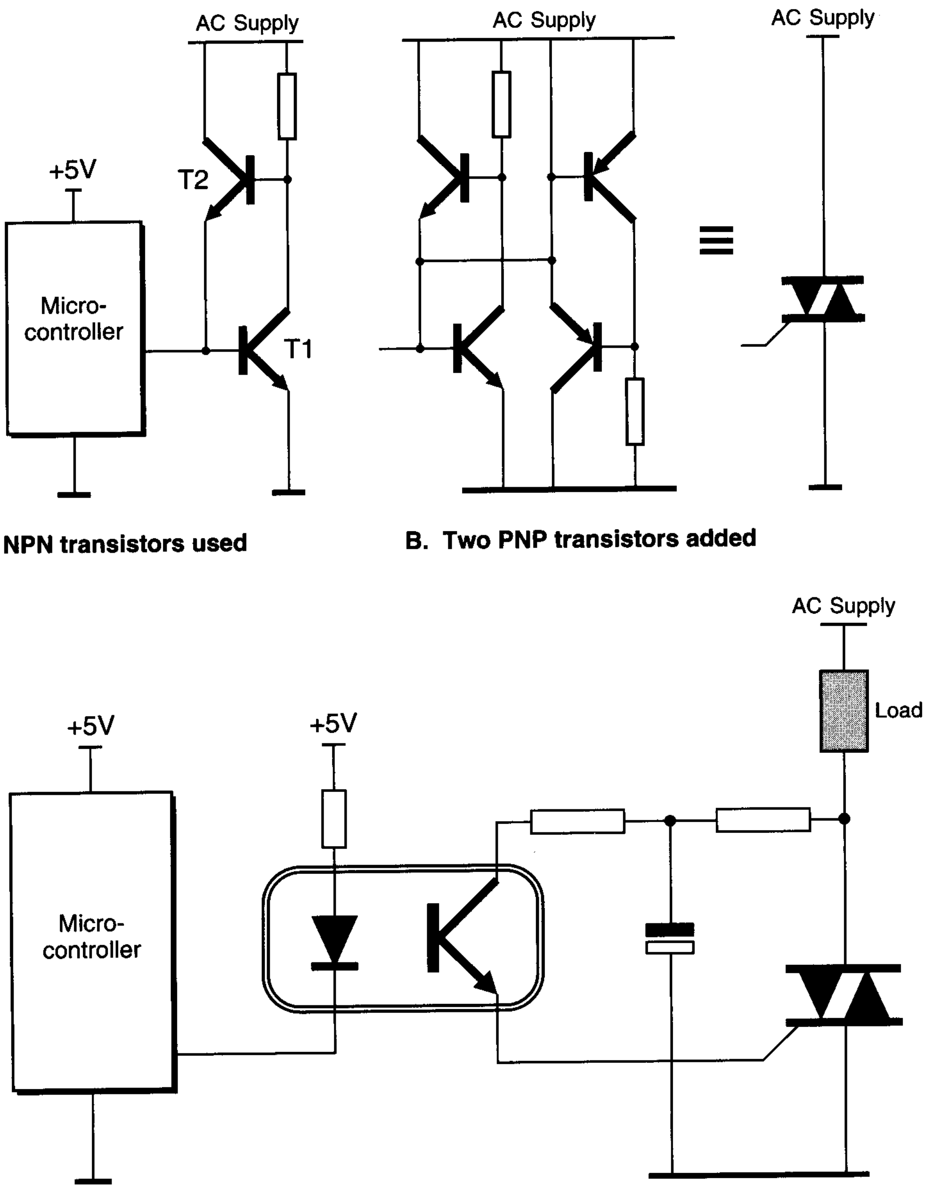


Figure 10-9. Continuous power control

Consider the transistor circuit shown in part A in Figure 10-10. When the microcontroller turns transistor T1 ON, by driving a HIGH output, transistor T2 will also turn on. Transistor T2 will now maintain transistor T1 ON even when the microcontroller stops driving a HIGH. Or, to put it another way, the microcontroller needs to supply only a pulse to turn **on** the transistors. The only way to turn **off** these two cross-coupled transistors is to reduce the supply voltage to 0. An AC supply does this every half-cycle so the transistors will stop conducting at the next zero-crossing.

The two NPN transistors used in A in Figure 10-10 will control the positive half-cycles of the AC supply. We also want to control the negative-going half-cycle of the AC line supply, so we add two PNP transistors (B in Figure 10-10). The four-transistor combination has proved so useful to control AC circuits that it is fabricated as a single device and called a TRIAC. Rather than connect the TRIAC gate directly to the microcontroller output, it is preferable to use an optoisolator (C in Figure 10-10).



A. Two NPN transistors used

B. Two PNP transistors added

C. Optoisolator added

Figure 10-10. Full-cycle AC control using a TRIAC

To have full control over the power delivered to an AC load, the microcontroller need detect only a zero-crossing of the AC power circuit, wait a calculated time, and then pulse the TRIAC on. Figure 10-11 shows a USB-controlled lighting panel. The application on the PC host has six "intensity" slider controls that supply data to a USB lighting panel controller that determines when each of six TRIACs should be pulsed. Using the framework we created in Chapter 6, I took less than a day to design, code, build, and test this application. For the full source code and schematics, see the Chapter 10/Lighting Panel directory on the CD-ROM.

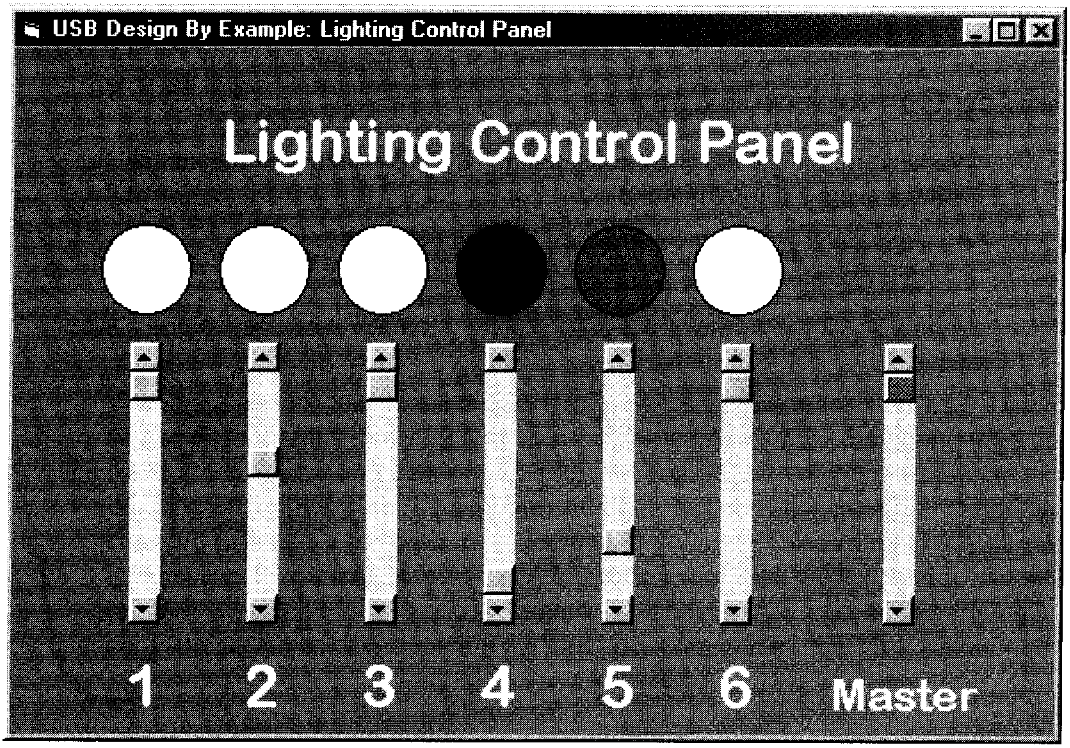


Figure 10-11. A lighting control panel

REAL-WORLD ANALOG SIGNALS

Thus far in this chapter, we have discussed two simple methods of generating an analog output using digital pulses:

- The pulse width modulation method produces an analog voltage equivalent to the integrated **average** voltage of the pulses; this technique is useful for slowly changing analog voltages.
- The delayed-pulse method of triggering a TRIAC was successful because our supply voltage was an analog function.

The next section discusses creating an arbitrary analog waveform.

Analog Conversion Examples

We need to understand two fundamental variables when creating an analog voltage using a digital method:

- Accuracy—how much error can be tolerated in the signal
- Speed—how fast the signals need to be

Figure 10-12 shows the effect of increasing accuracy when generating a sine wave.

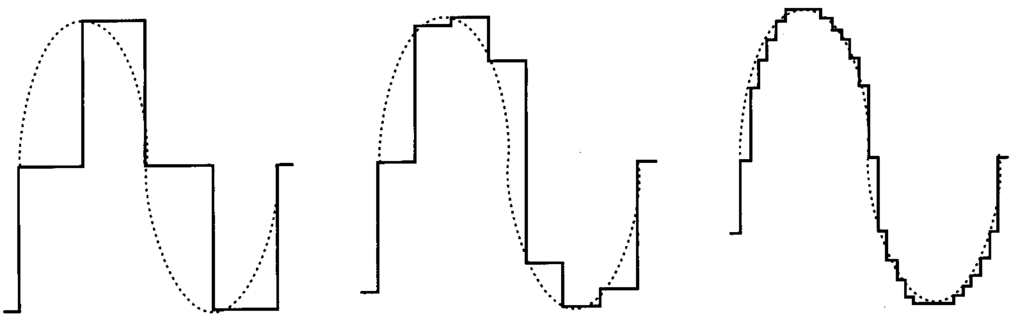


Figure 10-12. Generating an accurate analog waveform

The difference between the maximum signal and the minimum signal is divided into steps—the smaller the steps, the more accurate the output signal will be—but the cost also increases with accuracy. 8-bit digital-to-analog converters (DACs) are common, but converters of 12, 18, and 24 bits are also available. An 8-bit DAC can create 256 discrete steps from a reference voltage. Figure 10-13 shows a typical 8-bit DAC.

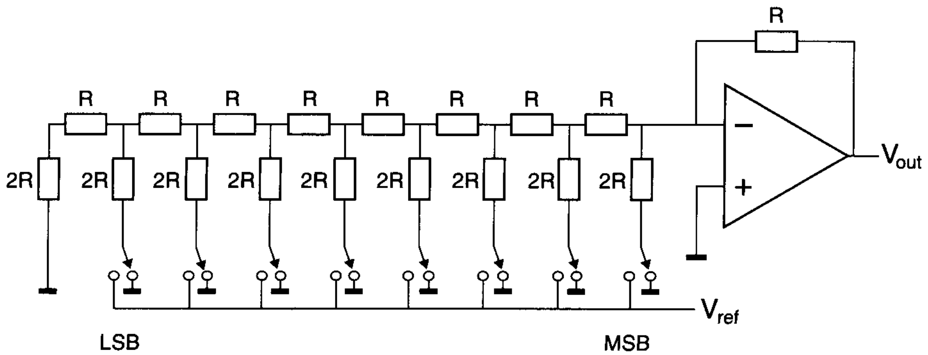


Figure 10-13. An R-2R ladder DAC

The digital inputs control the connection of current-steering resistors to a VREF source or to ground. Starting at the least significant bit of the digital input, each resistor has a value twice that of the previous source, with the exact value set by the ratio of two resistors. The outputs of all of the current sources that are on are summed to produce the desired analog output voltage. This is a relatively simple architecture to implement, assuming the resistors for each current source can be properly adjusted to the necessary precision. The resistors in commercial DAC components are usually thin-film and are laser-trimmed to their final value.

The speed at which the microcontroller can calculate and output a new value will determine the maximum frequency that this technique can generate. Analog amplifiers are used to create the desired output voltage from the DAC output.

Of several methods for reading an analog voltage, the choice depends on the highest frequency that needs to be captured (Figure 10-14).

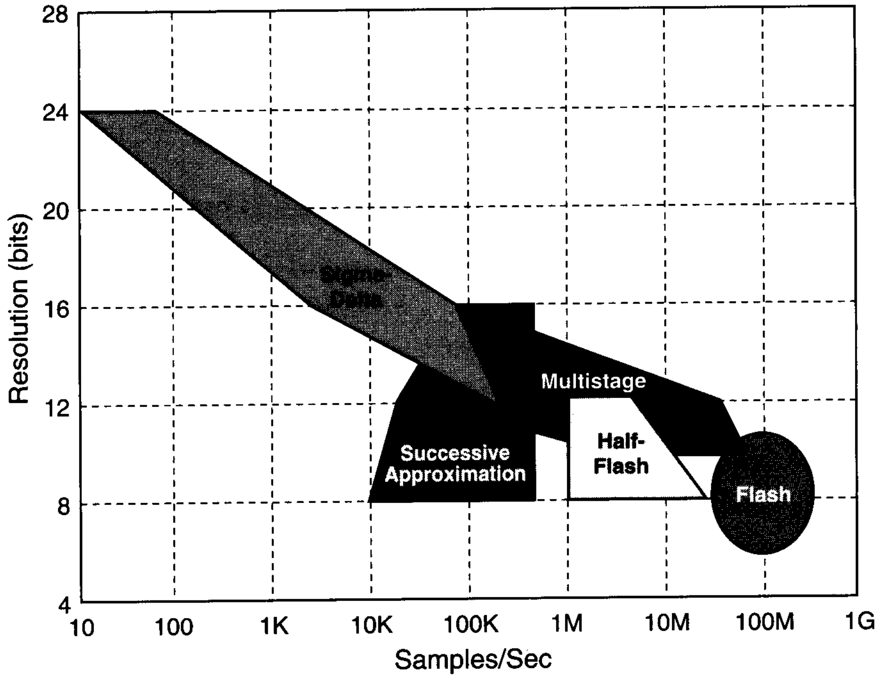


Figure 10-14. A range of analog-to-digital converter (ADC) solutions is available

Very high frequency waveforms are captured with a flash converter (Figure 10-15)—this essentially consists of *n* comparators with an increasing fraction of VREF on their input pins. The analog input is applied to all comparators whose outputs indicate if the input voltage is greater or smaller than their reference voltage. A digital output is extracted from this array.

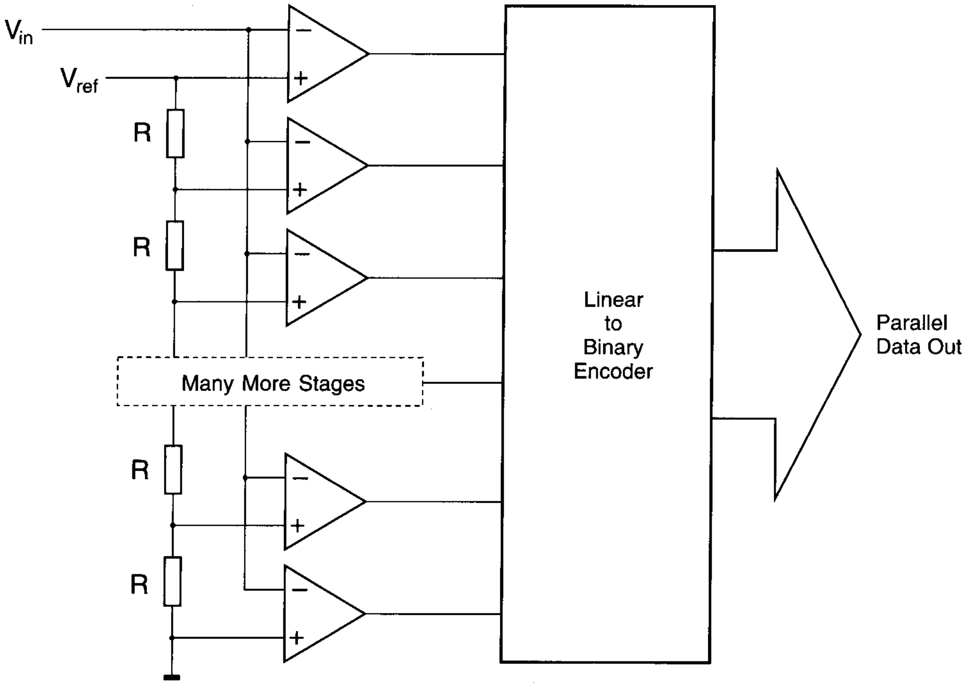


Figure 10-15. A high-speed flash analog-to-digital converter

A variation of the flash, the half-flash, or subranging converter, makes two passes at the signal—an initial "coarse" quantization, which is subtracted from the input signal, and a second "fine" quantization of what remains. Half-flash takes up less space on an IC and still offers relatively high speeds; however, a sample-and-hold circuit will now be required at the input.

A successive approximation ADC uses a DAC to create a voltage and compares this voltage with the input voltage. A binary search is used to hone in on the input voltage (Figure 10-16). The input voltage should remain stable throughout the resolution phase through use of a sample-and-hold circuit.

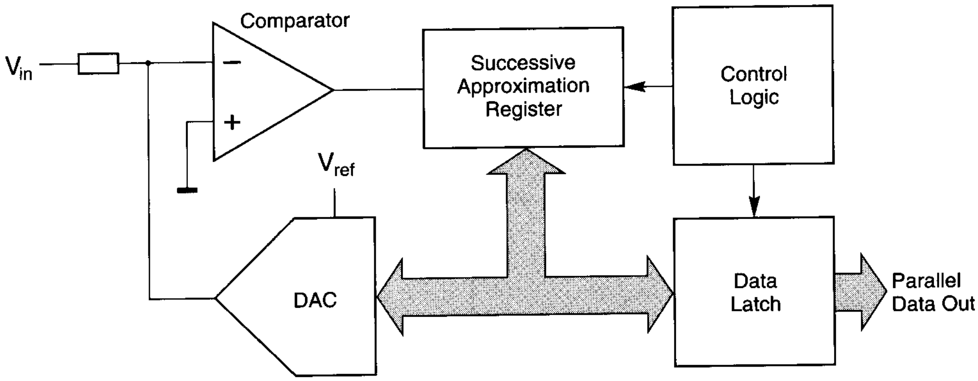


Figure 10-16. A successive approximation ADC

A sigma-delta DAC is a modern converter design made possible by advances in digital, rather than analog, integrated circuit manufacturing. The hardware is similar to a successive approximation analog-to-digital converter (ADC) but has an added integrator (Figure 10-17). The input signal is oversampled, typically at a rate 64 times the maximum input frequency, and these samples are integrated to produce a Difference signal. This is fed back to be summed with the input signal, and the cycle repeats with the bit-stream generated closely tracking the input signal. The 64x oversampling means that sample-and-hold circuitry is not required at the input, and steep antialiasing filters are not required at the outputs.

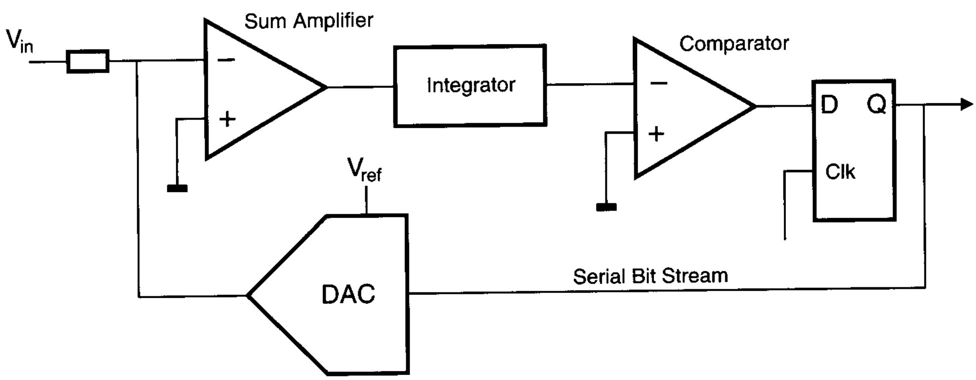


Figure 10-17. A sigma-delta ADC

It is possible to build both the successive approximation DAC and sigma-delta DAC using precision resistors and operational amplifiers and let a microcontroller run the control loop. But I do not recommend this approach. Instead, I recommend using integrated, tested DAC and ADC components that are readily available from a variety of suppliers such as Analog Devices, supplier of DAC and ADC components and Burr-Brown.

I recommend that the interested reader refer to *Digital Signal Processing and the Microcontroller* by Grover and Deller (ISBN 0-13-081348-6) for a comprehensive study of analog-to-digital and digital-to-analog converters.

Sensor Inputs

Now that we know how to get an analog voltage into the PC host using USB, we can look at various sources of analog voltages. A sensor, also called a transducer, creates an electrical signal because of some physical change. The signal can be generated because of a change in resistance, capacitance, or inductance, and we can use various techniques to detect and amplify these signals.

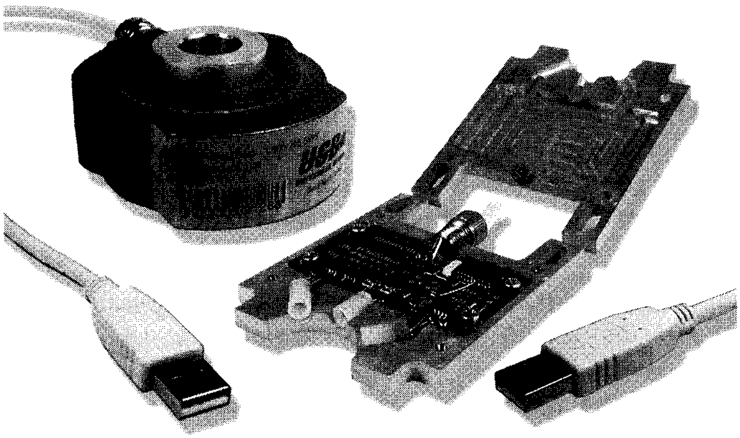
The range of sensors/transducers is as varied as the physical property that we are trying to measure. Table 10-1 lists just a few of the possible devices to consider.

Table 10-1. Physical attributes that could be measured

Angle or displacement
Angular acceleration/deceleration
Displacement and proximity
Extension or compression
Flow in gas, liquid, or solid
Humidity, absolute or relative
Linear acceleration/deceleration
Position, absolute or relative
Pressure in gas or liquid or sound
Temperature, absolute or relative
Torque or strain
Velocity
Weight and volume

For help in deciding on the correct transducer and signal conditioning, see the very useful *Transducer Handbook* by H.B. Boyle (ISBN 0-1506-1194-4). Bob Boyle has spent most of his career working with all types of transducers, and he is able to recommend how to solve many difficult problems. The book also includes an extensive supplier's directory to get your project started on the right foot.

An innovative approach to sensors is being taken by Hohner Corporation (Figure 10-18). All of the required sensor electronics are integrated directly in the sensor, which then interfaces directly to USB.



Courtesy of Hohner Corp.

Figure 10-18. Integrated shaft encoder and surface sensor from Hohner

Hohner's shaft encoder operates in two modes depending on a selection in the software driver. The encoder can report absolute position information or speed information with up to 24 bits of accuracy. The range will vary according to your application, and this too is programmable. All control and measurement is actually done by the USB microcontroller inside the sensor, so there is no concern about the non-real-time nature of the Windows operating system. The shaft encoder is supplied with a graphing demonstration program, hooks to enable it to input data directly into an Excel spreadsheet, and an interface to enable it to operate within a LabVIEW environment (to be described in the next section).

The Hohner surface sensor is an optical noncontact device that can measure characteristics such as roughness, reflection index, and color. Because the device is noncontact, it allows measurements of soft, liquid, or gel surfaces in difficult-to-reach places. The USB microcontroller uses a single light source and

measures the reflected light distribution at two sensors. This light distribution data is collected in real-time and reported to the PC host on request. The unit can also be calibrated to support a wide range of applications. Software is included with the surface sensor, which displays graphical data or can supply information to a data acquisition program.

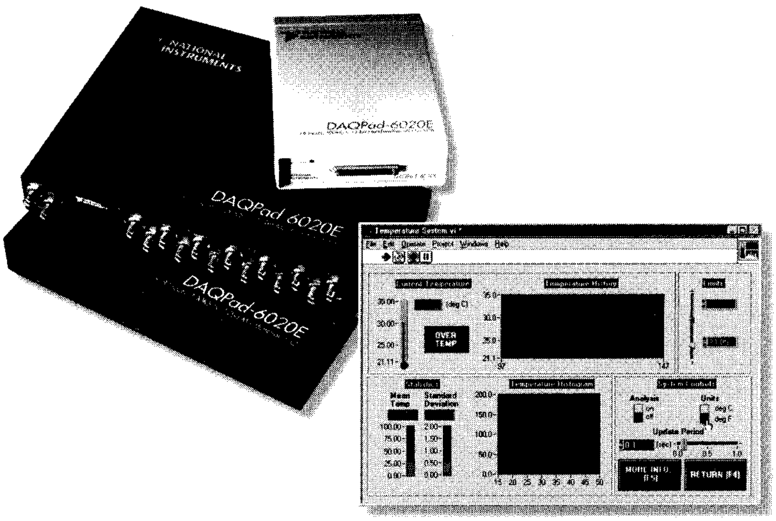
Hohner Corporation is extending its range of USB sensors to meet customer requirements. As this book was being written, Hohner was introducing several new devices, including a USB spectrometer and a USB ellipsometer (used for measuring the polarization of light). For datasheets, see the Chapter 10/Hohner directory on the CD-ROM.

DATA ACQUISITION AND INSTRUMENTATION

Perhaps you do not want to build your own real-world analog and digital input and output modules—you just want to buy and use them. This section discusses two approaches: a USB module-based series from National Instruments and an Industrial Control Base from Granite Microsystems.

USB Module Examples

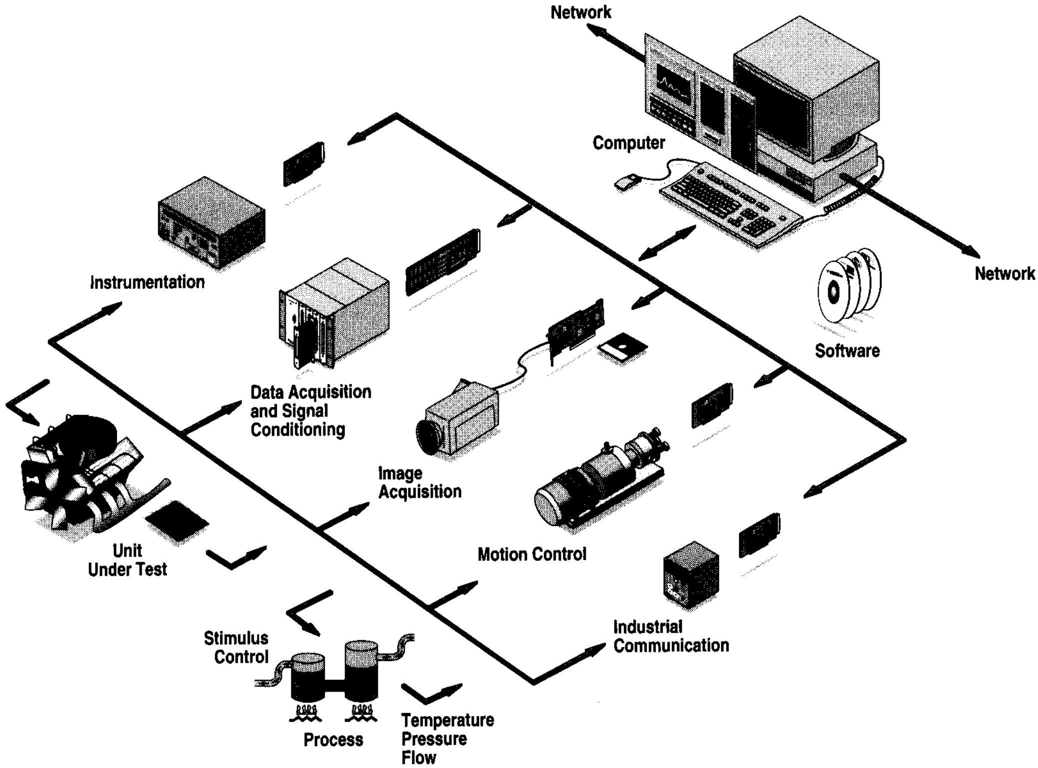
Figure 10-19 shows some National Instruments industrial modules and a screen shot of LabVIEW, a Windows-based application that simplifies the management and operation of data acquisition and instrumentation applications.



Courtesy of National Instruments.

Figure 10-19. Industrial USB-based data acquisition modules

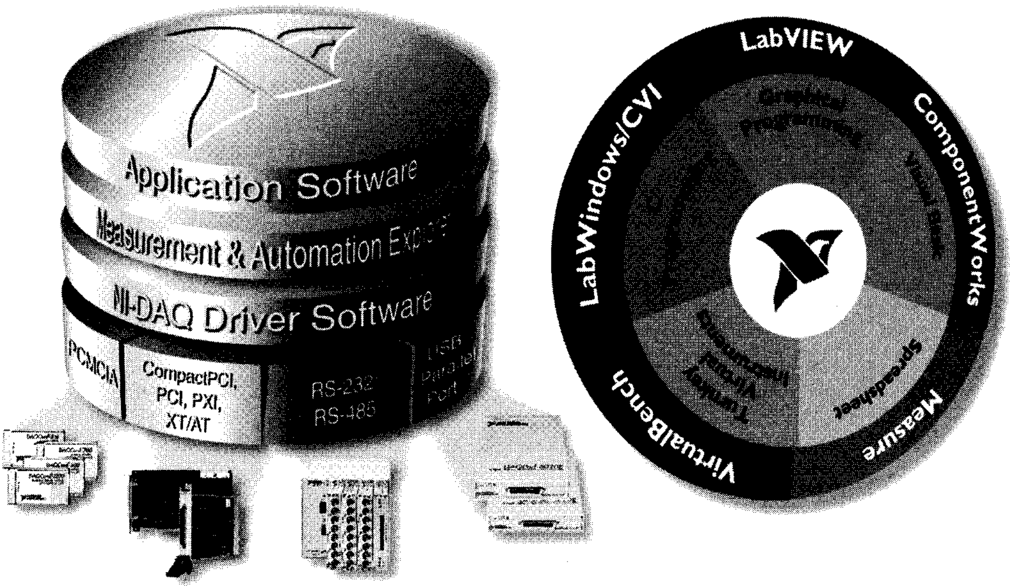
National Instruments has built up a large range of data acquisition equipment with the PC as the controlling element (Figure 10-20). The company has recently added a USB-based set of instrumentation that is proving popular in smaller or portable installations.



Courtesy of National Instruments.

Figure 10-20. Large range of data acquisition products

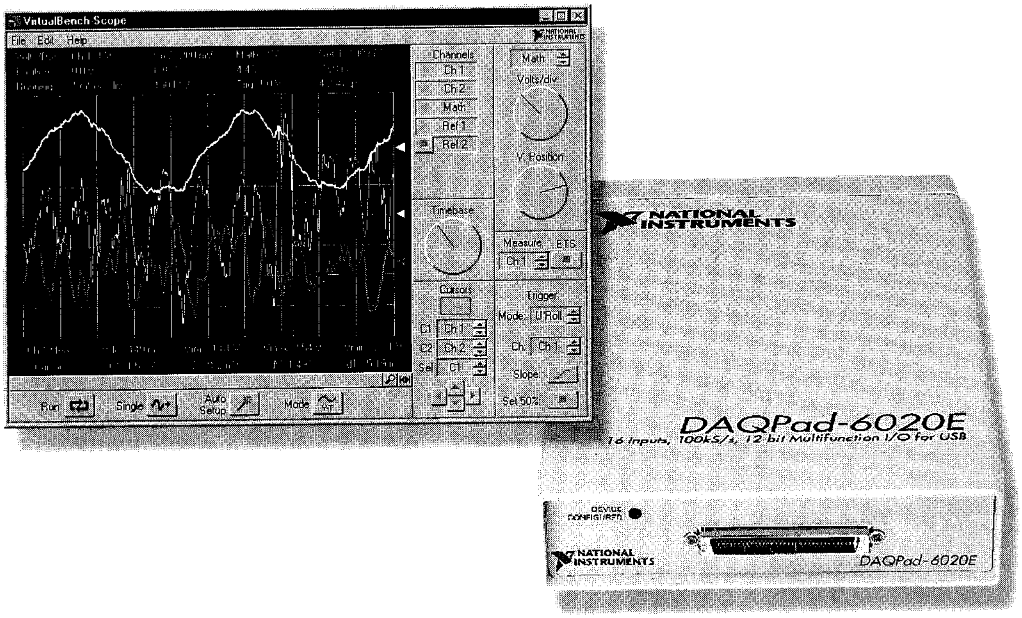
The heart of the National Instruments product line is an extensive software driver base called NI-DAQ. This measurement and automation software manages the low-level communications to all of the data acquisition hardware. When a USB data acquisition product is added to the PC, its enumeration directs the driver to NI-DAQ. The device is then fully integrated into the measurement and automation system. All instruments are controlled from the PC host using visual-based software (Figure 10-21).



Courtesy of National Instruments

Figure 10-21. Visual control of instrumentation

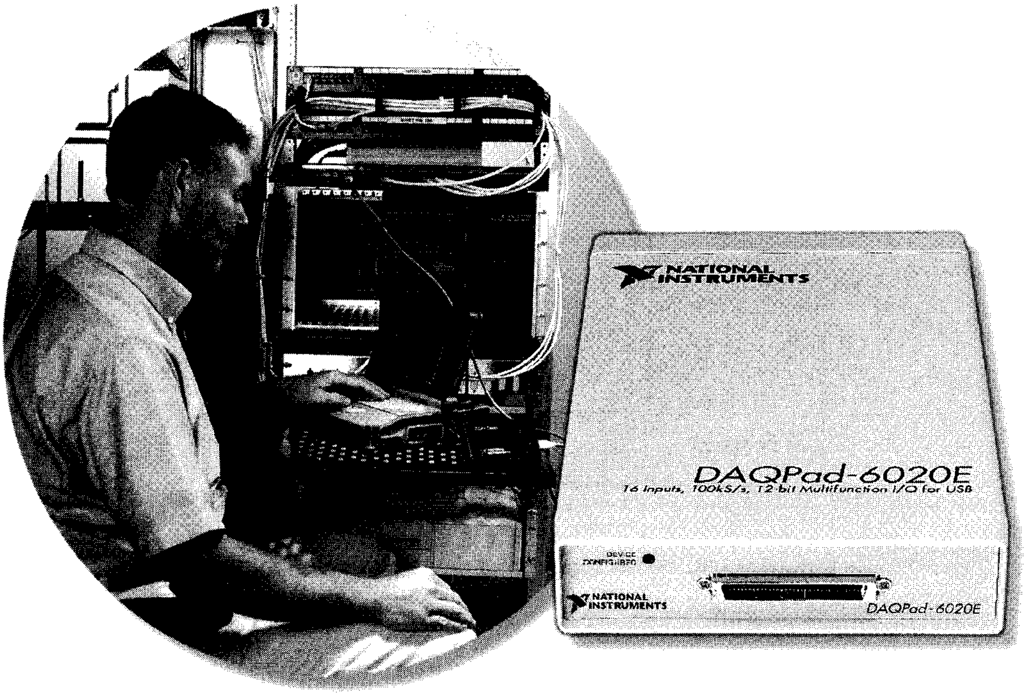
Different applications are available to provide a range of features. LabVIEW, for example, includes an array of building blocks used to model the experiment or data acquisition project you are working on. The data is collected from the real world and displayed graphically at the PC host. I was particularly impressed with the Virtual Instruments software that is a set of prewritten LabVIEW programs that operate as instruments. Figure 10-22 shows a virtual oscilloscope that will operate with the USB-based hardware also shown.



Courtesy of National Instruments.

Figure 10-22. An oscilloscope is software plus hardware

All of National Instruments USB-based products are built using the enclosed, portable design shown in Figure 10-22. This makes them particularly useful for on-site data acquisition. The equipment under test is connected to signal conditioning circuits as shown in Figure 10-23. The signal conditioning can provide isolation, amplification, attenuation, or whatever is required to preprocess the signals for instrumentation. The operator arrives with a laptop computer, connects to the USB data acquisition equipment, and takes measurements; the operator then returns to the lab to analyze the data.



Courtesy of National Instruments.

Figure 10-23. USB enables walk-up instrumentation

Inside each USB-based instrument is the now familiar block diagram shown in Figure 10-24. The microcontroller manages the data communications to the PC host and controls the analog and digital data acquisition circuitry that connects, via an external signal conditioning unit, to the equipment under test.

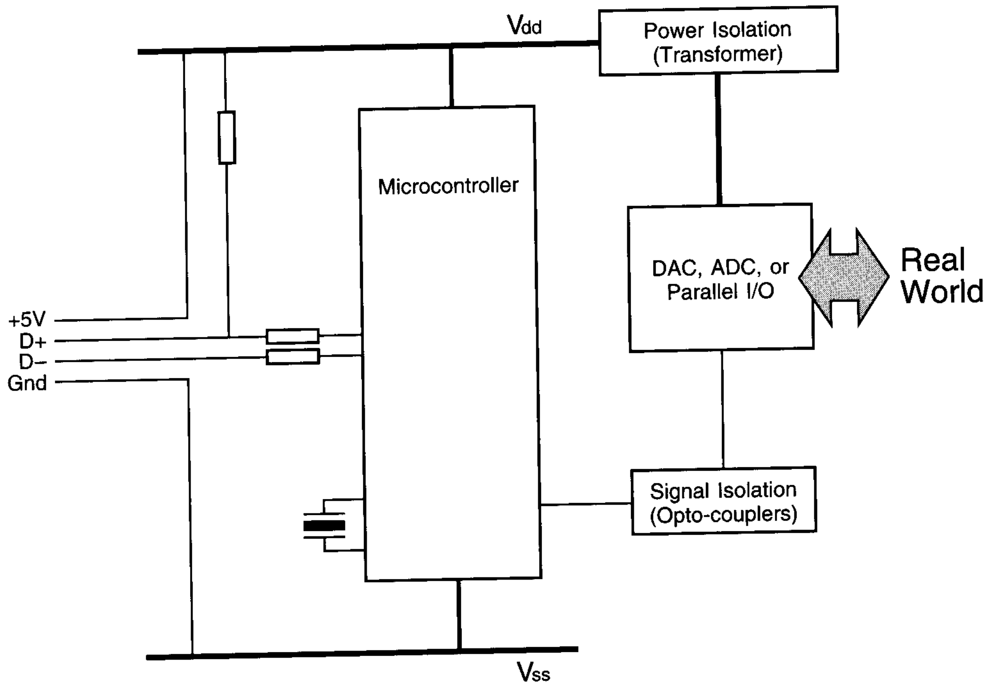
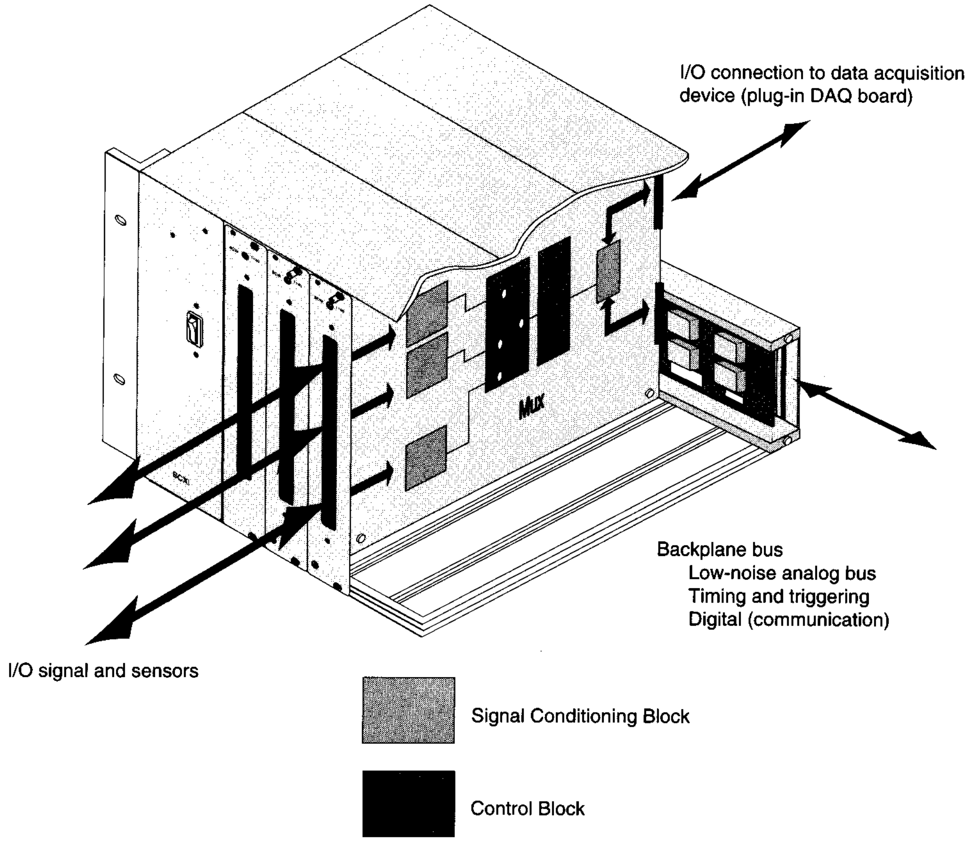


Figure 10-24. Block diagram of data acquisition module

The USB solution can successfully monitor 32 to 48 points at a sample rate of 100 kHz. If more points are required, then multiplexer circuitry can be added—National Instruments has a wide range. In typical applications this circuitry is attached to the equipment under test, and the data acquisition equipment (Figure 10-25) is connected later. Multiplexing will, of course, reduce the sample rate.



Courtesy of National Instruments.

Figure 10-25. Multipoint, high-speed data acquisition system

Many of the available instrument racks communicate using the GPIB protocol (General Purpose Instrumentation Bus). This solution first appeared on Hewlett Packard equipment in the 1990s and has now been adopted by most of the instrument manufacturers. National Instruments provides a USB-to-GPIB bridge that instantly enables a USB-equipped laptop to manage and control this setup (Figure 10-26).

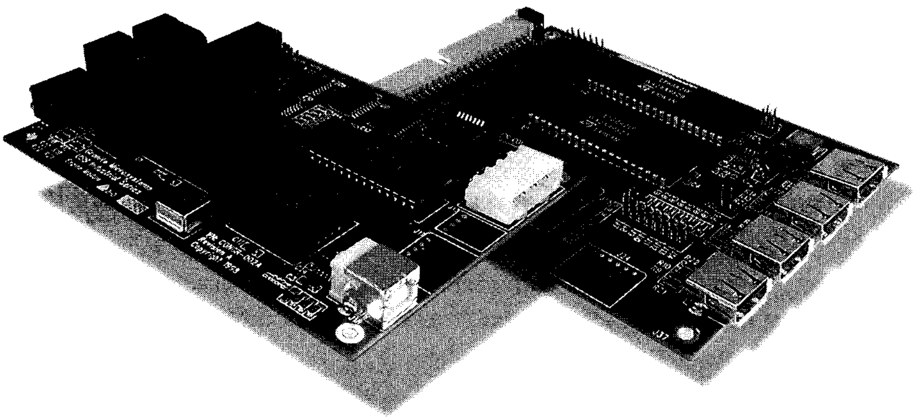


Courtesy of National Instruments.

Figure 10-26. USB-to-GPIB controller

USB Industrial Series Example

Granite Microsystems has expanded its expertise in industrial control by adding a USB Industrial Series line to its current expansive line of products. The first two products are an engine module and an IO-24 module that can be attached to the engine module (Figure 10-27). Figure 10-28 shows the block diagrams of each of these modules.



Courtesy of Granite Microsystems.

Figure 10-27. USB industrial series from Granite Microsystems

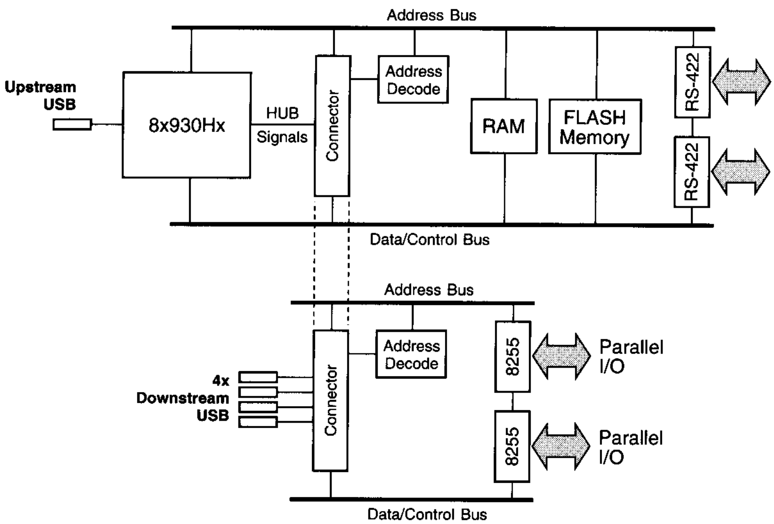


Figure 10-28. Block diagrams of Granite modules

The engine module is designed around the Intel 8x930AX, although the 8x930HX, with its four hub connections, can also be used. The physical hub connections are located on the IO-24 module. A large installation would have multiple engine modules fed by USB cables from an IO-24 hub module, which would have a single USB cable back to the PC host. Individual sensors such as those from Hohner or the USB modules from National Instruments could also be attached to the IO-24 hub board.

Each engine module uses flash memory, and Granite Microsystems supplies a utility that enables the microcontroller firmware to be field-upgraded.

The expansion connector is a key part of the design and enables multiple stackable modules. The engine module also contains two RS-422 connectors for serial expansion and eight uncommitted status LEDs. The IO-24 module provides a 50-pin connector that is compatible with industry-standard I/O modules from Opto-22 and Grayhill I/O modules

Device Bay Example

No discussion of real-world I/O and USB would be complete without an introduction to device bay. Device bay is a mechanical standard designed around USB (and IEEE 1394) that allows multiple manufacturers to build interoperable modules.

Figure 10-29 shows the two approved form factors with the "condo" of most interest to this chapter.

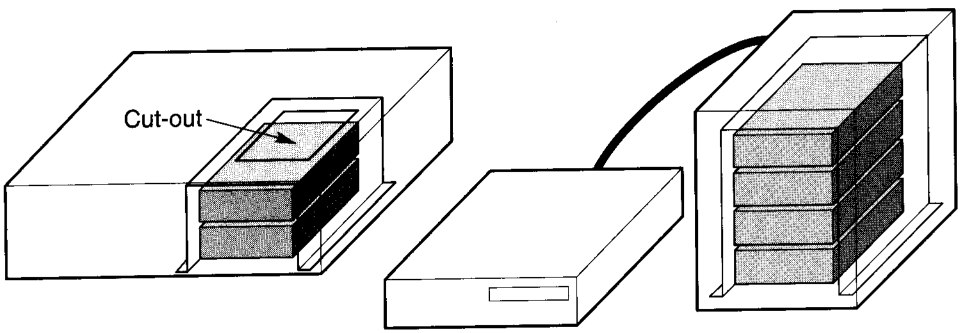


Figure 10-29. Device bay form factors

A device bay module is a fixed-size, metal box with hot-plug USB (and IEEE 1394) connectors on the back and device connections on the front. This format can be used in a PC environment to add such components as disk drives, DVD players, and broadcast tuners. It can also be used in data acquisition and process control applications where a configurable instrument is required. Note that device bay is a mechanical, form-factor standard—the electronics **inside** the module are the same as in the previous examples.

The benefit of the device bay format is that it makes the resulting equipment easier to use.

CHAPTER SUMMARY

The real world is a big place. Fortunately it is easy to interface almost any piece of electronics equipment and almost every type of sensor to USB. We discussed a variety of examples from lamps and motors, to analog signals and sensors, to three examples of data acquisition and process control. The versatility of USB has been amply demonstrated, and its intelligence to pre- and postprocess data for the PC host enables it to be used in the widest variety of applications. Software is available for the PC host to take advantage of current applications and is ready to tackle new applications as they arrive.

CHAPTER 11
I LIKE THE SOUND OF THAT

Audio on the PC has come a long way from the beeps and chirps of the first IBM PC. Sound was initially designed into the PC platform as an alerting mechanism, so the obnoxious noise was warranted. Today, however, the sound quality of the PC is comparable with high-end consumer audio equipment.

There is much talk currently about "digital audio," standards such as AC-97 and MPEG, but in fact we have had digital audio since 1983 when the Musical Instrument Digital Interface, or MIDI, was introduced. The growth of MIDI has been hampered by lack of standardization. Fortunately today, thanks mainly to the publication of the General MIDI System specification and its inclusion in the Windows operating system, MIDI is seeing widespread use in all aspects of the PC platform.

Before we look at specific implementations of audio on USB, it is useful to look at the different sound formats to understand why multiple industries are converging. First let's address the fundamental issue of why "go digital" at all. Typical sounds are generated in an analog way, and our ears hear in an analog way. So why convert to digital and back again? If the listener of an audio signal is close to the source of the audio signal, there is no need to go digital. However, if the audio source has to be distributed to the listener, then digital technology can be a big benefit. Traditional audio distribution systems are analog-based and include some amount of noise. Along the distribution path to the listener, this audio signal will pick up more noise. Once delivered to the listener, the signal and noise are not separable, so the received audio quality is lower than the source audio quality. The ratio of signal-to-noise has decreased.

A digital distribution system, a CD-ROM for example, stores an audio signal with a very high signal-to-noise ratio along with more error-check information. If noise is later added to this digital signal, it may be filtered out at the receiver using the digital check-bits. The received audio quality is the same as the source audio quality.

A digital approach to sound also enables more capabilities.

Once in a digital format, sound can be processed just like any other computer data type. We are no longer limited to two static channels, left and right. Audio separation can be added digitally. We can also implement 3-D positional sound.

The digital AC97 specification includes 5.1 digital channels: left front, center, right front, rear left, rear right, and special effects.

Before the advent of digital mixing, mixing multiple audio sources of different sampling rates and sizes was difficult. Digital mixing also allows all application programs to share the same sound generation hardware on the PC.

USB allows this sound generation hardware to be outside the PC chassis. There are many high-frequency clocks, disk drives, and switching-mode power supplies inside the modern PC—this is not a friendly environment for creating high-quality analog audio. USB allows the digital-to-analog converter (DAC) to be in a separate box at the far end of a USB cable. The improved fidelity is noticeable.

What we are seeing is a convergence of multiple audio sources into a single PC platform implementation. And USB is a catalyst making this happen.

CREATING DIGITAL SOUND

Digital sound files can be created using two basic methods: real-world sound can be sampled or a sound can be synthesized.

Sampled Sound

Figure 11-1 shows a typical analog waveform.

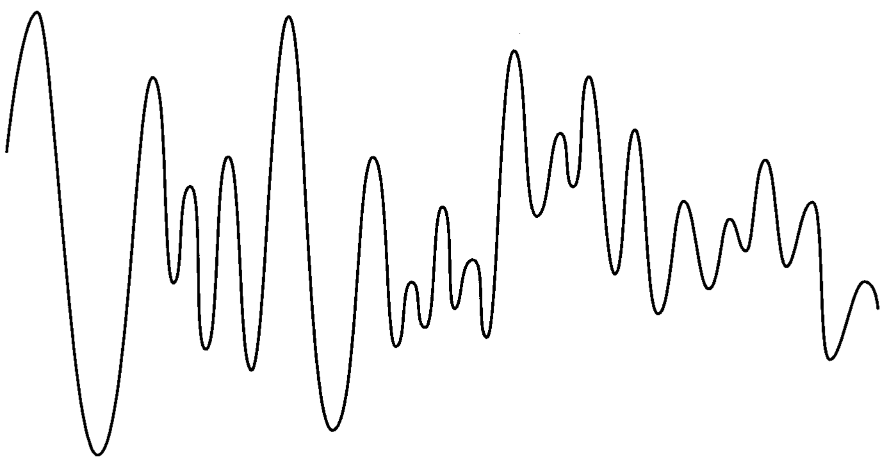


Figure 11-1. A typical analog waveform

The analog waveform is **sampled** by the PC using an analog-to-digital converter that periodically converts the instantaneous signal amplitude into a digital number. The sample period, or sample rate, is constant and must be at least twice the highest frequency within the incoming waveform. This sampling process is called Pulse Code Modulation, or PCM, and is the predominant method for storing uncompressed sounds in digital format. Windows .WAV files store PCM-coded information. Multiple industry-standard, signal-quality levels have been defined that vary in the accuracy of the sample and the sampling rate. Playback of a PCM-coded file requires sending the digital samples to a digital-to-analog converter at the same rate as they were sampled. Figure 11-2 shows a PCM audio subsystem overview.

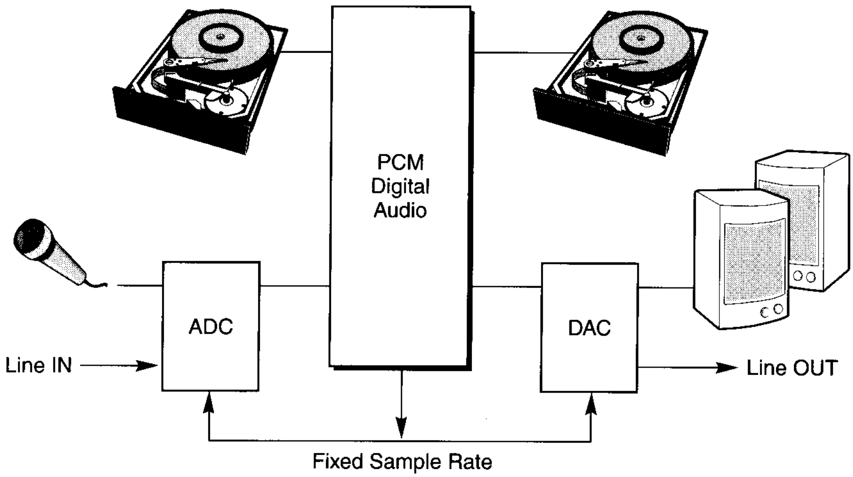


Figure 11-2. Overview of a PCM audio subsystem

The de facto standard in a PC platform for sample and playback has been the Sound Blaster series of add-in cards. Different models are available to cover the standard range of sound qualities. The programming interface to a Sound Blaster card is a range of I/O ports, so all application software is effectively "tied" to this hardware implementation. DOS applications write directly to these I/O ports with mutually exclusive access. Any alternate solution needs to be register-compatible with a Sound Blaster card so applications software would not change. Microsoft has now defined a software interface for ALL audio devices, and new applications should be written to this more portable interface. Microsoft's DirectX and WDM drivers make this legacy DOS hardware interface virtual and add mixing capabilities that enable multiple clients access to the sound hardware

(in much the same way as the Windowing GUI does for the display). Microsoft also includes a Sound Blaster Emulator with Windows 98; the emulator is a WDM driver that intercepts accesses to the range of I/O ports used by a Sound Blaster card and translates them into the new audio interface.

The Windows 98 audio driver can redirect sound files to a USB device that will implement the digital-to-analog conversion outside of the PC host. Later in the chapter we'll design one of these I/O devices.

Synthesized Sound

Figure 11-3 shows a waveform of a typical synthesized sound waveform. It has a start, a middle, and an end. The middle consists of four sections—the initial sound or attack, an early loss of initial amplitude or decay, a continuance of sound or sustain, and finally a release. The richness of sounds is described by this envelope; a simple on-off sine wave is not representative of real-world sounds.

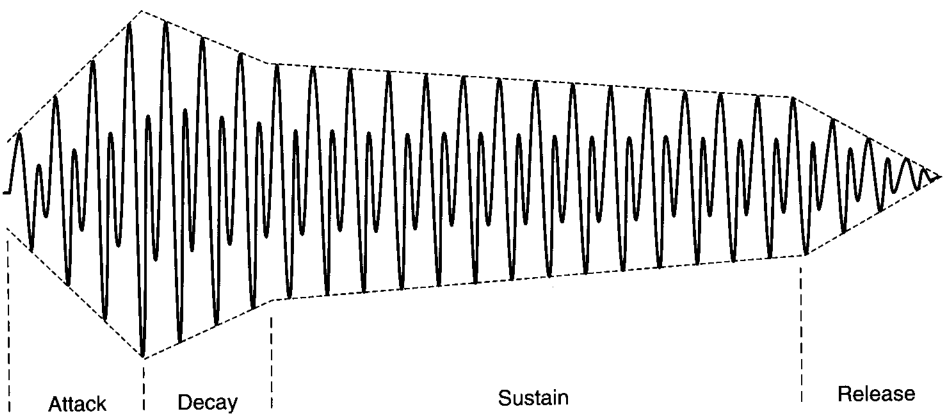


Figure 11-3. A typical synthesized sound waveform

The waveform in Figure 11-3 can be represented in two MIDI messages, each of which is only three bytes' worth of data! The MIDI protocol is a very efficient method of representing musical performance information. The de facto standard for MIDI was defined by the MPU 401 from the Roland Corporation (now Edirol). This was also an I/O port interface. A synthesizer is required to translate MIDI messages into sounds, as shown in Figure 11-4. Although the original synthesizer was implemented in hardware on a Sound Blaster board, this

is not required. The synthesizer can be implemented in hardware or software, inside or outside the PC host. The Windows operating system includes a hardware-independent device driver that allows different hardware implementations with no change to the application software.

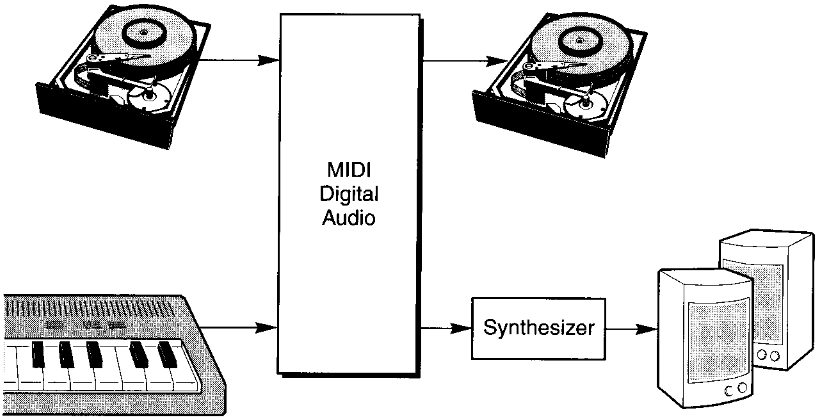


Figure 11-4. Overview of a MIDI audio subsystem

Looking deeper inside MIDI, we discover that there are actually four components: the MIDI communications protocol, the MIDI hardware interface, the MIDI synthesizer, and the MIDI stored file format.

MIDI Protocol

MIDI information is transmitted in MIDI messages, which can be thought of as instructions that tell the synthesizer how to play a piece of music. The synthesizer receiving the MIDI data generates the actual sounds. MIDI messages are transmitted as a serial bit stream at 31.25 Kbps with 10 bits transmitted per byte (a start bit, 8 data bits, and one stop bit). Why 31.25 Kbps? The original hardware divided the 8-MHz ISA bus clock signal by 256! The protocol divides the single physical MIDI channel into 16 logical channels that are controlled independently. A MIDI message is made up of a status byte that is generally followed by one or two data bytes. At the highest level, MIDI messages are classified as being either Channel Messages, which are targeted for a logical channel, or System Messages, which control the entire performance.

The MIDI protocol defines the activation of a particular note and the release of the same note as two separate events. For example, when a key is pressed on a MIDI keyboard, it transmits a **Note On** message on one preprogramed MIDI channel. The Note On status byte is followed by two data bytes, which specify key number (indicating which key was pressed) and velocity (how hard the key was pressed). When the key is released, the keyboard will send a **Note Off** message. The Note Off message also includes data bytes for the key number and the velocity.

At the beginning of a MIDI sequence, a system message is sent on each channel used in the performance to set up the appropriate instrument sound for each part. The General MIDI Specification includes the definition of a General MIDI Sound Set (a patch map), a General MIDI Percussion map (mapping of percussion sounds to note numbers), and a set of General MIDI Performance capabilities (number of voices, types of MIDI messages recognized, etc.). A MIDI sequence that complies with the General MIDI Specification will play correctly on any General MIDI synthesizer or sound module.

MIDI Hardware Interface

The first implementation of a MIDI interface on a PC platform used the game port. I don't know why—that would appear to be the worst choice, because this port was designed to interface to analog joysticks. The only valuable attributes I could discover for this port were that it was available and FREE—it was included on all early PC platforms, and few people used it. A lot of software, and processor bandwidth, is required to send and receive serial messages on this parallel port. Compared with the MIDI protocol, however, the processor has lots of time to do this, and if the PC platform isn't doing anything else, then there's no problem. But as people wanted the processor to do more things, the "resource-wasting" software had to be replaced by a serial controller, typically on the sound card.

There are two ways of implementing a MIDI interface cable: cheaply and correctly. The difference is shown in Figure 11-5: A cheap cable puts the signals on the correct pins, and the correct cable uses optoisolators and careful shield screening to eliminate "hum."

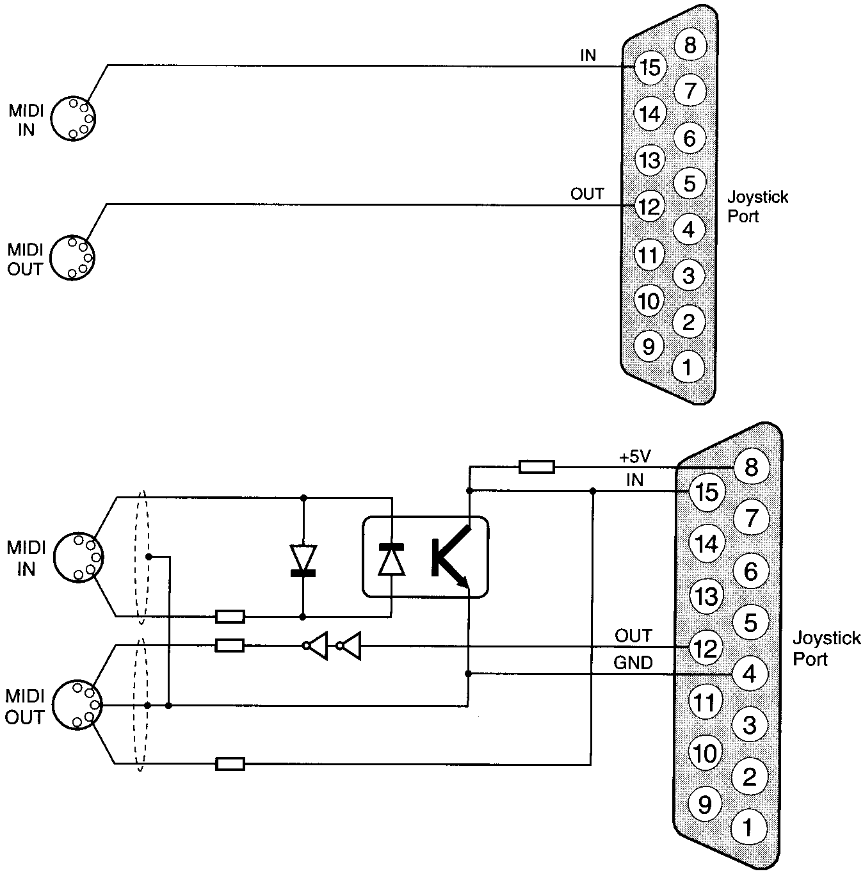
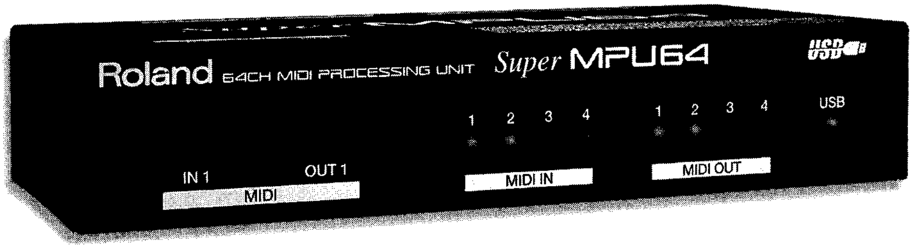


Figure 11-5. MIDI cables are cheap or correct

The modern PC does not have a joystick port—it was removed in the *PC 99 System Design Guide* because its burden on system performance was becoming excessive.

The modularity and layered implementation of the Windows operating system allows a MIDI application's system calls to the MIDI hardware to be intercepted and redirected. The Roland Corporation supplies a device driver that redirects MIDI requests to USB. A USB subsystem, called the S-MPU64, receives these requests and drives the real MIDI ports. The S-MPU64 includes 4 MIDI IN connections and 4 MIDI OUT connections (Figure 11-6). This entry-level system is powered by the USB port and provides the connectivity required for a range of MIDI equipment.



Courtesy of Edirol Corp.

Figure 11-6. USB-to-MIDI interface from Edirol (Roland)

Roland has recently introduced a "big brother" to the S-MPU64 that includes multiple audio connections. Roland included audio line input and output, and local guitar and microphone inputs, on their UA-100 product (Figure 11-7) to create a complete digital audio subsystem.



Courtesy of Edirol Corp.

Figure 11-7. USB-based audio subsystem from Edirol (Roland)

The Roland UA-100 enumerates as an I/O device with four separate functions. A block diagram (Figure 11-8) highlights the major elements connected to a single USB cable. The MIDI connection requires custom software, but later in this chapter we will design an audio subsystem similar to this solution.

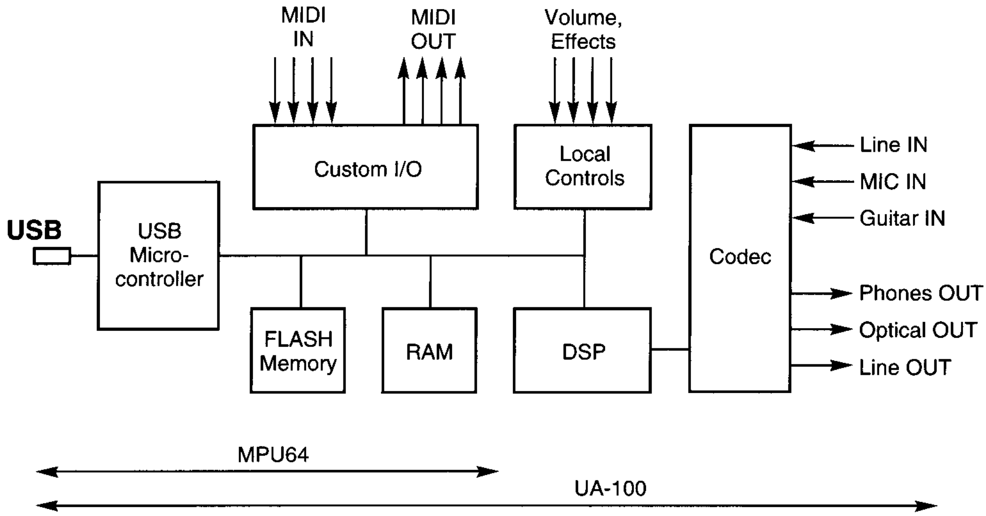


Figure 11-8. Block diagram of Edirol's audio subsystem

MIDI Synthesizer

Three methods are currently used to synthesize sounds: frequency modulation, wavetable, and physical modeling.

Frequency Modulation, or FM, synthesis involves combining fundamental sine waves of different frequencies and amplitudes to create a complex waveform. FM synthesis is a good technique for beeps, bangs, explosions, and space-type noises but is not suitable for creating the richness of a musical instrument.

Wavetable synthesis starts with actual music instruments; these are recorded, and short digitized sections are stored. When an application program requests a particular sound, a sample is retrieved, processed, played back, and typically looped. Wavetable sound quality is generally very good, provided the original recordings are of high quality.

Physical modeling is a combination of physics and art; a mathematical model that takes into account all of the forces on, say, a violin string, is used to predict the sound that plucking or bowing the string would make. The result is an incredibly lifelike reproduction, though the initial coding of each model is time-consuming.

The output of a synthesizer is sound; this could be analog sound in the case of a hardware implementation of a synthesizer, or it could be digital sound, high-resolution PCM format, for a software implementation. Hardware implementations have been popular in the past, but the increased performance of today's PC platform processor allows the processor to do a superior job for lower system costs.

MIDI File Format

MIDI data files are very small when compared with PCM data files (that is, .WAV files). For instance, PCM files require about 10 MB for each minute of CD-quality audio while a typical MIDI file might consume less than 10 KB for each minute. This reduction is possible because the MIDI file contains only the instructions required to re-create the sounds and not the sounds themselves. A synthesizer will generate the actual sounds from these instructions.

The International MIDI Association publishes a Standard MIDI Files Specification that defines how time-stamped MIDI data should be stored. The interested reader can purchase this specification from www.midi.org (they would not provide a copy for inclusion on the CD-ROM). This standard allows different applications such as sequencers, scoring packages, and multimedia presentation software to share MIDI data files.

USB's Support for Sound

USB was designed to support sound very well. The isochronous transfer type was specifically designed into USB to support audio and other time-dependent data transfers. Refer back to Chapter 2 to recap the attributes and system scheduling of isochronous transfers. The characteristics of this USB data type are similar to those of PCM digital audio.

USB transmits frames from the PC host at a 1-ms rate, and an isochronous I/O device will require data transmitted in **every** frame. For example, CD-quality sound at 16 bits (4 bytes for left and right channels) and a sample rate of 44.1 kHz will need 176.4 bytes of data transferred in a 1-ms frame. Because you can't practically transfer 0.4 bytes, the operating system sends nine frames of 176 bytes and then one frame of 180 bytes to maintain the 44.1-kHz sample rate. The I/O device would receive this data at the high USB bus speed, buffer it, and supply it at 44.1-kHz to the left- and right-channel DACs. Isochronous transfers are not allowed on a low-speed connection.

During enumeration an isochronous I/O device typically requests no USB bandwidth. Bandwidth will be allocated once the sound application is initialized and the device is configured. If the PC host has this bandwidth available, then the isochronous I/O device is enabled and is guaranteed to get the data requested at the data rate requested. If the PC host does not have the bandwidth available, it interrogates the isochronous I/O device to discover if it can operate at a lower bus bandwidth. The isochronous I/O device provides this information via **Alternate Configurations** that request less and less of the USB bandwidth. An isochronous I/O device is **required** to have a setting that uses no USB bandwidth—yes, this means that the device will not be operational, but the operating system is now able to give sensible error messages to the user and is still able to communicate with the device. This was preferred over leaving the isochronous device in an unconfigured state, which can potentially confuse the user— "I plugged it in, why isn't it working?"

Once enumerated, an isochronous I/O device provides or consumes data at its negotiated rate. If an error occurs on the data transmission, the data packet is not retried. Because the data is time-critical, late data is as useless as no data. The receiver typically uses the previously received data rather than leaving "a gap" and tries to resynchronize with the data source as soon as possible.

Example 1: Audio Output

Building USB speakers is very easy because almost all of the work is already done for us in the operating system and via the USB Specification. The bytes are already ordered exactly as they are needed to be sent to the DAC. One hardware wrinkle in the design is the availability of DACs for stereo audio applications. All of the available components are manufactured with a serial interface designed to operate gluelessly with a DSP, and they include ADC input channels (that is, they are coder/decoders—codecs). The interface is too fast for a microcontroller to "bit-bang." Parallel interface stereo audio DACs are available, but these components also include analog inputs and other mixing circuitry. So our choice is a serial codec (such as the AD1849) and a parallel-to-serial converter with a PAL controller, or a parallel codec (such as the AD1845). Figure 11-9 shows the hardware implementation using an Anchor Chips device and a serial codec.

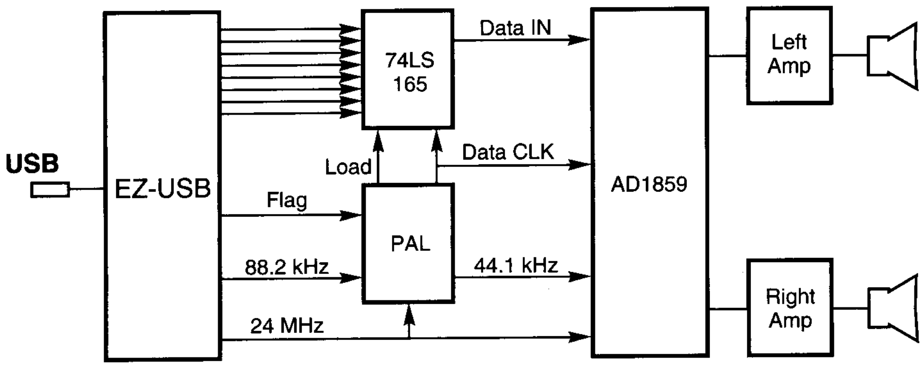


Figure 11-9. Discrete implementation of USB speakers

The Anchor Chips device supports double buffering on isochronous endpoints (see why it's called the EZ-USB?!), so our task is to empty one buffer while the other is being filled from USB. This example empties the buffer under interrupt control.

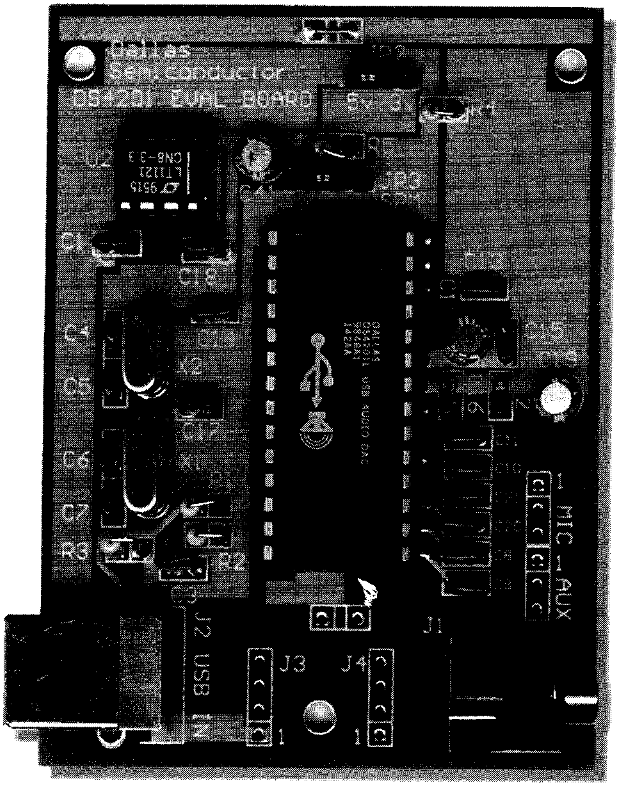
An internal timer is used to generate an 88.2-kHz sampling signal, and the overflow of this timer causes the EZ-USB to run an interrupt service routine. The microcontroller outputs the right-high byte, raises a flag, and then outputs the right-low byte. The flag causes the PAL to copy the high byte into the parallel-to-serial converter and to start the serial data transfer into the AD1849.

At the next interrupt, the microcontroller outputs the left-high byte, clears the flag, and then outputs the left-low byte. This interrupt processing consumes less than 20 percent of the processing time of the Anchor Chips device, giving it scope to implement other tasks. The complete code and equations for the parallel-to-serial control PAL can be found in the Chapter 11/Speakers directory on the CD-ROM.

No software needs to be written for the PC host, because the required support is already embedded in the operating system.

An applications note that implements the same design for an Intel 8x930AX microcontroller and a parallel DAC is also included in the Chapter 11/Speakers directory. This note contains more background and timing information.

If you don't plan on using the "spare" processing power of the microcontroller, then a simpler solution is available from Dallas Semiconductor (Figure 11-10).



Courtesy of Dallas Semiconductor Corp.

Figure 11-10. Single-chip implementation of USB speakers

Dallas Semiconductor has integrated the USB transceivers, the SIE, the control, and the DACs onto a single component (Figure 11-11). The microcontroller firmware is not available to change as it is in the previous example—the traditional tradeoff between flexibility and simplicity. The Dallas DS4201 also includes analog inputs that may be mixed with the digital audio from the PC, but these features are not used in this first example.

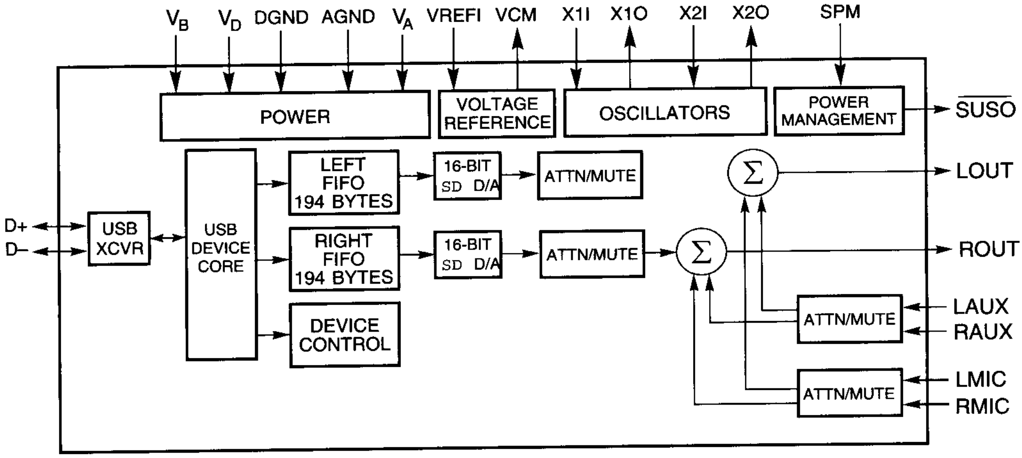
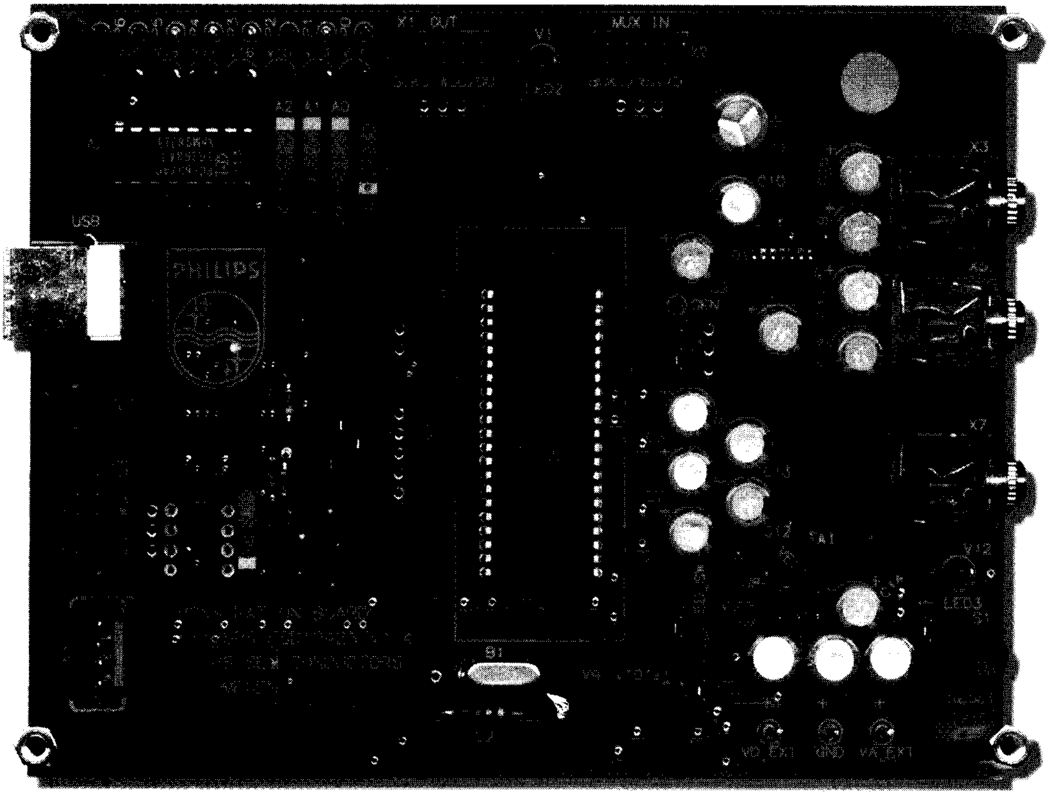


Figure 11-11. Block diagram of the DS4201

Example 2: Audio Input

The next example also includes audio input capabilities. Figure 11-12 shows a photograph of the demonstration board, and Figure 11-13 shows the block diagram of a Philips UDA1335 Audio Playback and Recording Peripheral. Dedicated hardware does all of the low-level USB and audio handling, and an internal MCS51 microcontroller implements the higher-level USB protocols. The UDA1335 is ROM-based, and the program is available to OEMs under license. Most users will find the customization "hooks" adequate for their applications.



Courtesy of Philips Semiconductors.

Figure 11-12. Philips UDA1335 demonstration board

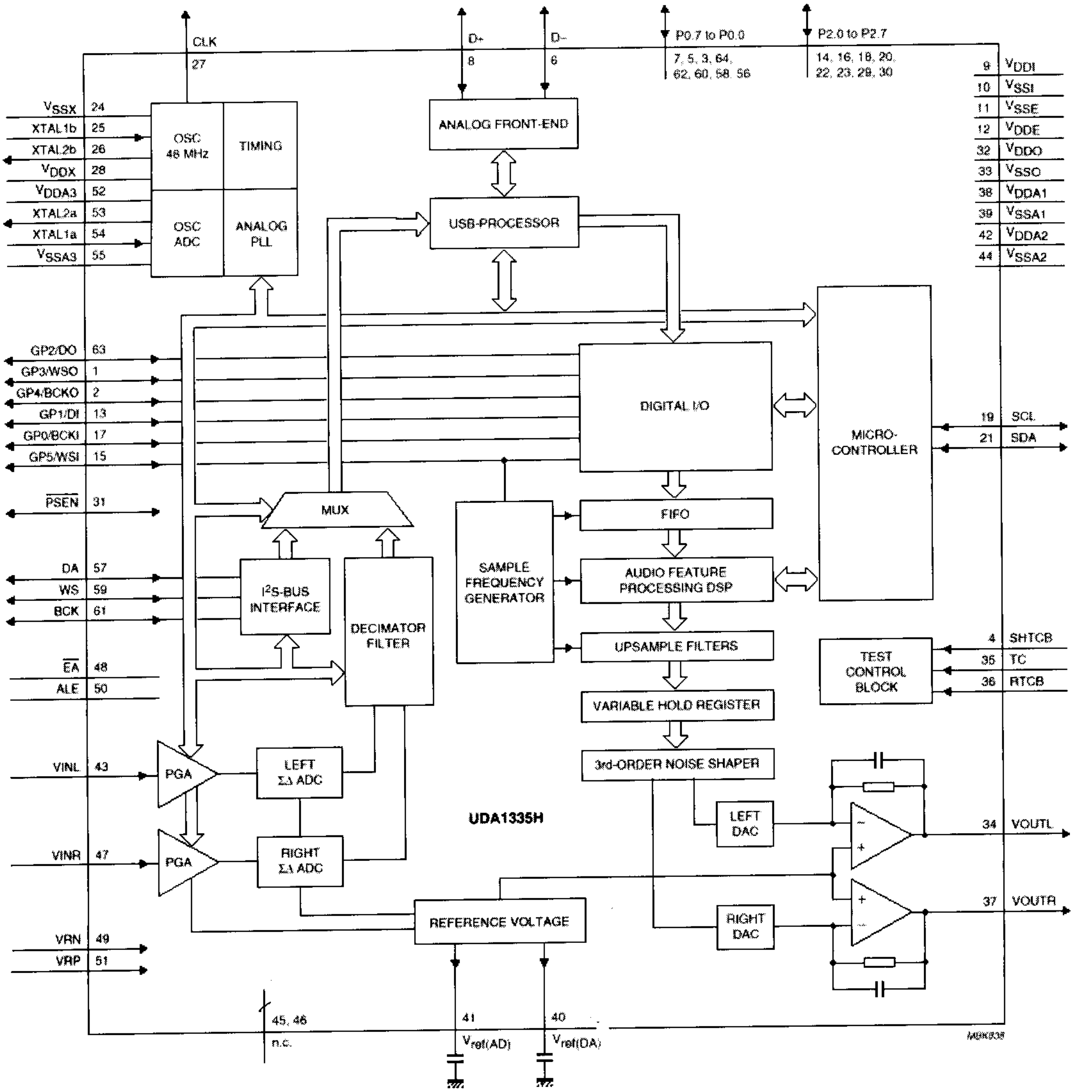


Figure 11-13. Block diagram of UDA1335 APRP

The UDA1335 supports four typical configurations via selection inputs and a custom configuration via an I^{2}C EEPROM. The EEPROM is also used to change the vendor ID and USB strings. Because most users will want their own identification strings, Philips makes it easy to program the EEPROM by providing a CODEC Configuration Editor program (supplied in the Chapter 11/Philips directory on the CD-ROM).

The I^2C bus is also used to enable more inputs and outputs. This is useful if you need buttons and/or lights on your peripheral device. The HID report technique we learned in Chapter 6 is used to access these buttons and lights—we just add an HID descriptor along with the many other descriptors in this example.

The UDA1335 has more features worth pointing out. It supports a 24-bit audio data format for higher fidelity. It includes software-controlled volume, treble, and bass via a Feature Unit. The device also provides a connection to an Inter-IC Sound, or I2S, bus.

The I2S bus was developed by Philips to efficiently move digital sound data between components. It is a 3-wire serial bus consisting of a data line for two time-multiplexed data channels (left and right), a word select line, and a clock line. A Digital Signal Processor may be connected to this bus to provide more local capability if required.

Function Control by Software

Our USB-to-LineOut example contains only AUDIOSTREAMING descriptors, which describe the isochronous data rates. We did not use the additional analog inputs on the device. A more sophisticated example would use these features and would want them controlled via software. USB also defines a set of AUDIOCONTROL descriptors that standardize the control of a generic audio device. Application software such as Cakewalk's suite and Sonic Foundry's Sound Forge use these interfaces.

Audio Control descriptors are fully described in "USB Device Class Definition for Audio Devices," which is included in the "USB Documentation" directory on the CD-ROM. The following section gives an introduction to the basic elements.

The requirement to describe physical entities such as signal inputs, signal outputs, selector switches, mixers, and other controls created a software model that defined these entities as standard **terminals** or **units**. A connected collection of terminals and units can be used to describe any audio device. A descriptor is defined for each terminal and unit type.

An **Input Terminal** describes the source of an audio input signal; this could be a "real-world" analog signal or a digital audio input stream.

An **Output Terminal** describes the destination of an audio output signal; this could be a "real-world" analog signal or a digital audio output stream.

A **Mixer Unit** transforms a number of input channels into a number of output channels (typically only one output channel), while a **Selector Unit** directs one of several input channels to an output channel.

A **Feature Unit** provides basic control over the incoming logical channel. It provides features such as volume, mute, tone, and graphic equalization. More sophisticated controls are implemented by a **Processing Unit, Dolby Prologic Unit, Stereo Extender Unit, Reverberation Unit,** or other units. Figure 11-14 shows the building block icons used to model the physical audio entity; Figure 11-15 shows an example using all of the features of the Dallas DS4201.

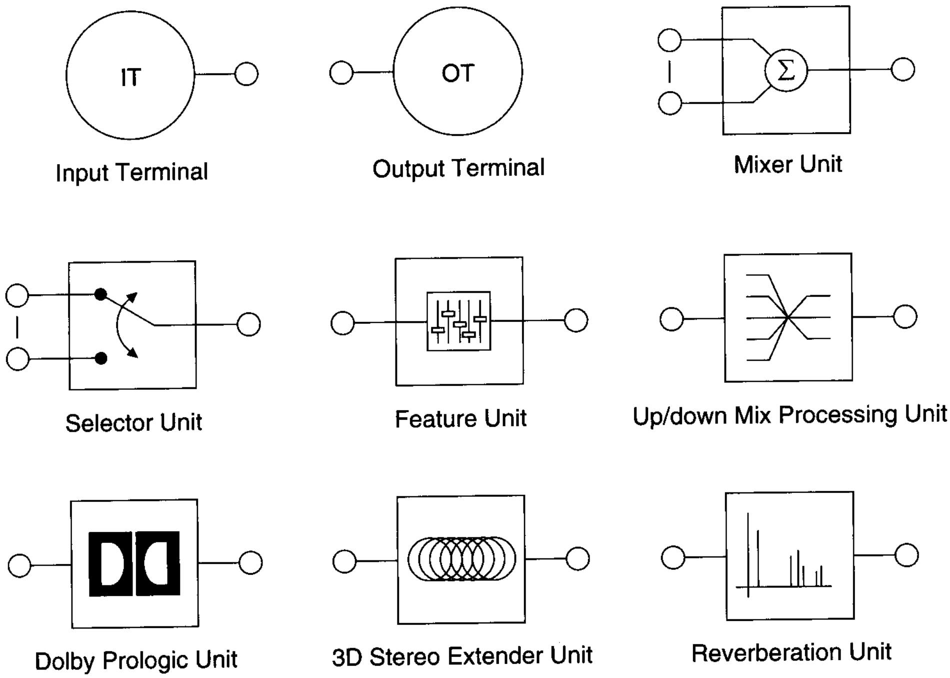


Figure 11-14. Icons used to model an audio device

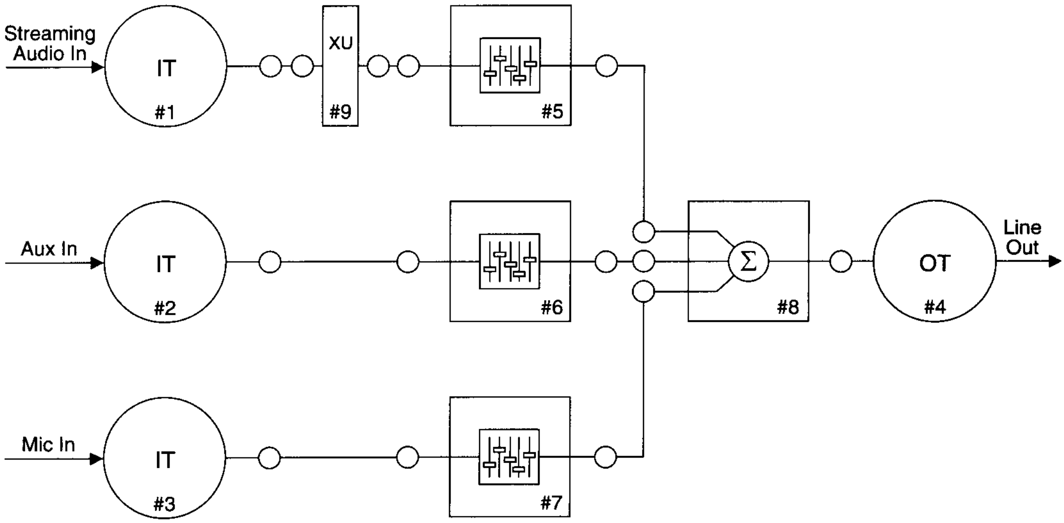


Figure 11-15. Software model of the DS4201

After uniquely identifying each of the terminals and units, a descriptor is written for each item and declared alongside the other USB descriptors. The full set of descriptors for the DS4201 is included on the CD-ROM under Chapter 11/Dallas Semiconductor/DS4201 Descriptors.pdf.

Following enumeration of this audio class device, it can be accessed and controlled by audio-based application software.

Before leaving the world of high-end audio equipment, I'll mention some products that use these, or similar, components in the configurations described. Figure 11-16 shows two USB-to-audio peripherals that make getting audio in and out of the PC easy, with high quality. These peripherals also support fiber optic–based interequipment connections for minimum system noise. I recently purchased a Pioneer Audio CD-ROM Jukebox and was impressed to discover a fiber optic output connector. I can directly import 24-bit sounds into my PC for distribution around the whole house.



Courtesy of Edirol Corp., Opcode Systems, Inc., and Jazz Hipster Corp.

Figure 11-16. Examples of commercial digital-audio equipment

THE TELEPHONE CONNECTION

The uniform treatment of digital sound within the USB architecture makes it easy to build an I/O device that is a combination of buttons, lights, and sounds. The telephone has been the number one tool for increasing business productivity for a long time. USB makes it easy to integrate it with the number two device—the PC platform. Together they can create a tool that should exceed our productivity expectations.

To get us in the right frame of mind, please don't think of this as adding a telephone to a PC computer. This will limit our imagination to obvious conclusions such as "well, the PC could do the dialing, I suppose." It is much better to think that you are adding a powerful PC to a telephone—what more could the telephone do if it had a 500-MHz Pentium III processor, 64-MB memory, and a 6-GB disk inside it?

Possibilities for an improved telephone now start to open up. Today's PC can easily do speech recognition, so saying "computer, dial Bob" is deliverable today. Figure 11-17 shows a USB-based business telephone from Nortel—the Meridian 9617.



Courtesy of Northern Telecom Ltd.

Figure 11-17. USB telephone from Nortel

Nortel includes Personal Voice Dialer software that implements the speaker-independent, voice-controlled dialing with their telephone. Once you've used this speaker phone, you'll wonder why you've been pushing those dialing buttons for so many years! The Meridian 9617 is a two-line business phone and includes advanced features such as call forwarding and conferencing, which can be controlled via Nortel's drop-and-drag Call Manager software. The phone makes extensive use of Caller ID for call tracking and recording and can even announce an incoming call while you're busy in an application—"it's your boss, you should take this call."

A "home" or "small-business" telephone would also include a PC-based answering machine. A software-based solution is more flexible and extensible; it could have multiple mail boxes—some public ones for callers to leave messages for specific people and some "password-protected" ones for the home owners to leave messages for each other or to activate home-automation tasks such as sprinklers or recording a TV program. The greeting played could be based on the time-of-day and implement a "do-not-disturb" feature to eliminate those 2:00 a.m. wrong-number calls. Automatic out-dialing could be used to distribute key messages to a group of people—"the little league game is canceled because the other team's bus broke down, see you at Monday's 4:30 practice."

Integration of the telephone and PC platform can create some very compelling products. Let's look at the design of a USB-based telephone so we can build similar, or maybe better, capabilities into our I/O device.

Example 3: Telephone Design

Figure 11-18 shows the design problem of a USB-based telephone divided into elements that we already have solutions for. The Data Access Arrangement (DAA) will vary from phone system to phone system, and there are many restrictions relating to phone-line connected equipment. These regulations vary by country; local documents can be obtained that define the scope and testing of a particular telephone authority interface.

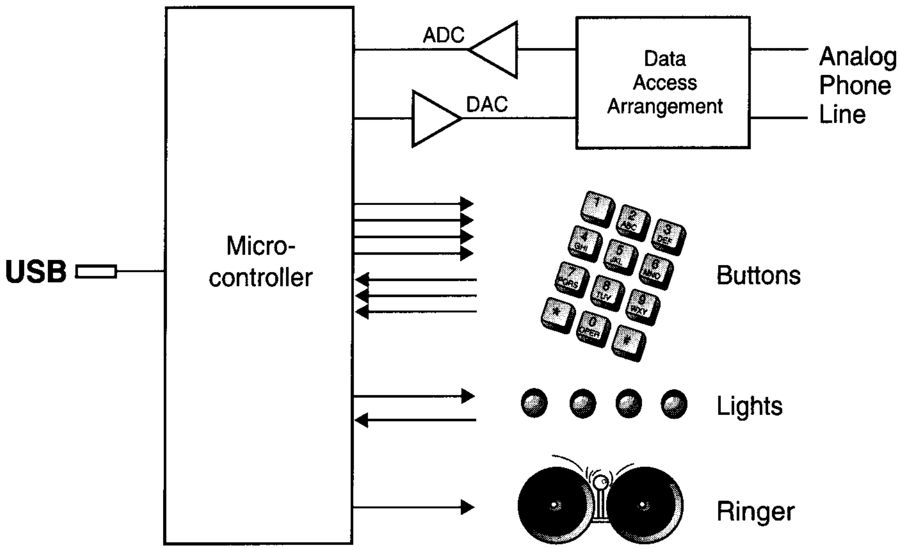
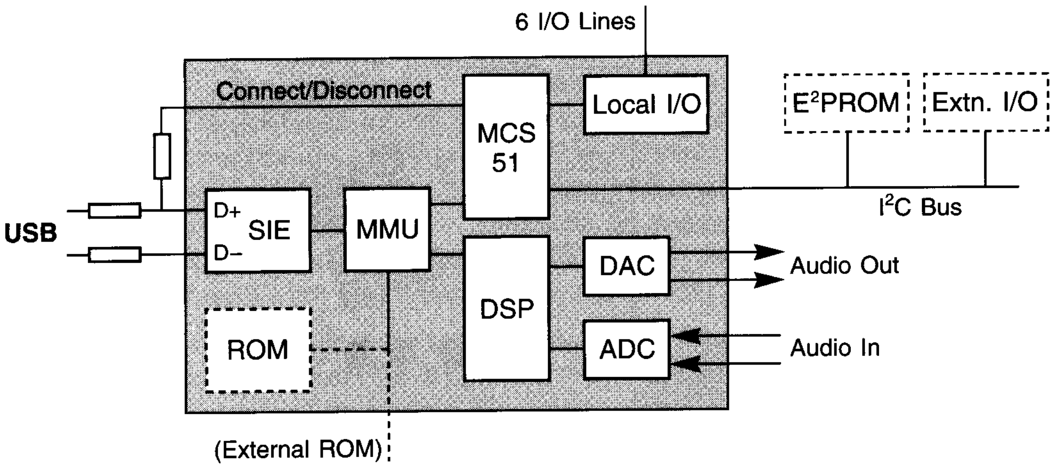


Figure 11-18. Major elements of USB-based telephone

A basic telephone could be designed around the Philips UDA1335 shown in Figure 11-12. More circuitry to generate and detect the local telephone company's signaling scheme would need to be added. In the U.S., a dual-frequency-based data transfer scheme (DTMF) is used with standard, flow-control tones (such as dial and busy), and components are available to implement this.

A speaker phone uses extensive digital signal processing, such as echo-cancellation and noise reduction, and these features require more processing at the PC host or via a local DSP. Figure 11-19 shows a component derived from the Philips UDA1335 that includes an embedded 50-MIP DSP with many of the DSP algorithms already implemented. This DA010 from ZMM was designed specifically to meet the needs of transferring processed audio signals between the PC host and the real world. Adding capabilities to the telephone, such as fax, modem, voice recognition, and speech compression, is now a simple matter of adding software. Designed primarily for telephone applications (analog, ISDN, and cellular), the component is also used in interactive toys.



Courtesy of ZMM.

Figure 11-19. USB-audio component from ZMM includes DSP

The ZMM DSP+Codec part can implement DTMF dialing and detection and all of the telephone flow-control signals in software, so no added hardware is required. A software-based fax modem could also be implemented with the support of host-based drivers (see the modem section of Chapter 8 for a discussion of the host-based processing approach). ZMM produces an evaluation board for OEMs (Figure 11-20).

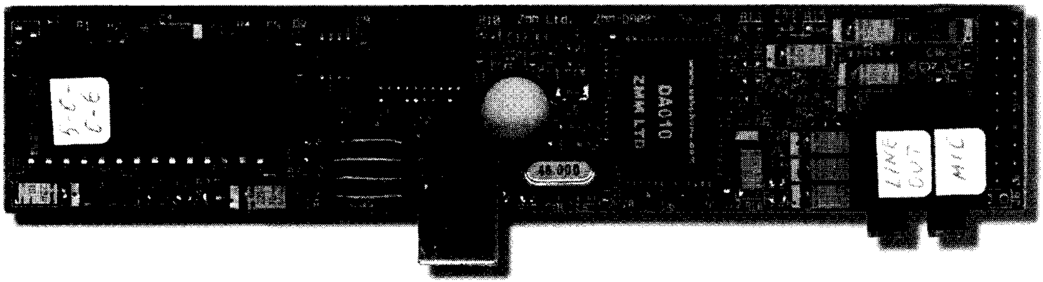


Figure 11-20. Evaluation board for ZMM's USB DSP CODEC

CHAPTER SUMMARY

The features built into USB enable it to support sound as easily as it supports other data types. Sound uses the isochronous capability of USB to implement real-time data delivery. The operating system already includes all of the software drivers required to implement a high-quality, digital audio system—this means that we have no PC host software to write (always a good sign). An audio design task consists of implementing the descriptors that define the device and its features, and writing a service subroutine for the isochronous interrupts. Alternately, one of the prebuilt audio devices could be used (see Appendix A for a full list).

The modular, building-block structure of USB allows sound to be easily added to any USB I/O device.

For More Information

Sound Blaster: The Official Book, by Ridge, Golden, Luk, and Sindorf. McGraw-Hill, 1994. ISBN 0-07-882000-6

A Programmer's Guide to Sound, by Tim Kientzle. Addison-Wesley, 1997. ISBN 0-201-41972-6

The General MIDI Specification can be obtained via www.midi.org.

Software-based answering machine: *DialTone* from www.sharkmm.com.

CHAPTER 12
I CAN SEE YOU

As far as USB is concerned, digital video is just like digital audio except there is a lot more data! Digital video uses the same isochronous features of the USB architecture, so there is no new theory to learn. It is time, however, to look at the amount of real data that we can put into a USB frame. Digital video will start to push the bandwidth limits of USB, so a good understanding of the actual packet timing is required to successfully implement a digital video I/O device.

We know that USB transmits packets of data at 12 Mbps in 1-ms frames. The following discussion will use the variable "Byte Times per Frame," or BTF, to identify how much data throughput can be expected. For a full theoretical analysis, see Chapter 12/USB Bandwidth Analysis on the CD-ROM. A summary of this paper is given here.

The maximum raw number of BTF is (12 Mbps / 1-ms) / 8bits/byte = 1500. We must reduce this number by the protocol overhead required to manage the link. Specific sources of overhead include packet organization, Start-Of-Frame and End-Of-Frame signaling, clock adjustment, and time reserved for control transfers. This reduces the useful BTF to about 1300.

Many of our early examples transferred small amounts of data between the PC host and the I/O device. Interrupt packets were used to guarantee delivery, so the transactions consisted of {Setup-Data INorOUT-Handshake} sequences. At full speed, a one-byte data packet is equivalent to 12 BFT, and at low speed this is (12*8) + (2 for preamble) = 98 BTF. Even at low speed this is a small load on USB, and in fact the actual load is much smaller than this because these interrupt packets are not sent every frame.

Our CD-quality audio example in Chapter 11 required isochronous packets with nine frames at 176 bytes and one frame at 180 bytes to maintain a 44.1-kHz sample rate with 16-bit samples. With 10 BTF of isochronous transaction protocol overhead per frame, this gives us a maximum of 190 BTF. Still a small load for USB. We have enough bandwidth for (1300 / 190)—almost seven separate CD-quality stereo audio channels on USB!

SIZING VIDEO DATA

This chapter deals with video, and one of the first things you realize about video data is that there is a lot of it. An NTSC TV screen, for example, is 720 x 480 x 24 bit = 1.0368 MBytes of data (a PAL TV screen is 720 x 576 x 24 bit = 1.244 MBytes of data). A TV picture is updated at 59.96 fields per second (two fields per frame), which results in a raw data rate of 31 MBps or 20,750 BTF. Because this **far** exceeds the maximum BTF of USB, it is easy to conclude that we cannot support full-screen, full-frame rate on USB. But let's see what some signal processing can do!

There are some easy choices we could make to reduce the data rate:

- Take fewer samples. A video signal is characterized by its YUV components (Y = luminance or brightness of the image, UV = chrominance or color depth of the image), and the 24 bits assumes full data capture (YUV = 4:4:4). The human vision system has poor color acuity, so it is possible to subsample the U and V components without the viewer noticing. By sampling both chrominance components at half the rate horizontally (YUV = 4:2:2), the sample size is reduced to 16 bits. By also sampling at half the rate vertically (YUV = 4:2:0), the sample size is reduced to 12 bits.
- Reduce the frame size. Industry standard sizes include CIF (352x288), QCIF (176x144) , and "small" (160x120). Figure 12-1 shows the resulting frame rate if we limit the maximum BTF for the video source to 1000.
- Reduce the frame rate. Flicker will be noticeable at frame rates below 15 fps and slow-motion effects will be evident at frame rates below 2 fps.

CIF	352 x 288 x 2 = 203 KB/frame	0.5 fps
QCIF	176 x 144 x 2 = 51 KB/frame	2 fps
Small	160 x 120 x 2 = 38.4 KB/frame	2.6 fps

Figure 12-1. Resulting frame rates at 1000 BTF

This scheme of transmitting raw video on USB does not produce very good results, and I should mention that a video camera operating at 1000 BTF would not be considered a good citizen. Names such as "data hog" come to mind. A single I/O device using this much of the USB bandwidth should have a WARNING label on the front because it will impact the operation of all other USB devices. If you must use an I/O device in this way, an acceptable solution is to install another USB root hub in the PC host. Each root hub can support 12 Mbps, so installing a second root hub would effectively produce 24 Mbps. Early USB video cameras operated this way and prompted many users to add another USB root hub just for the video camera.

Video Compression Is Essential

A better solution for reducing the data rate from the video source is to **compress** the video data, transmit the compressed data on USB, and then decompress the data on the PC host for display. A modern PC host has ample processor performance to decompress all styles of video encoding in software, so no cost is incurred at the host. Today's second-generation USB cameras use codecs and compression techniques that make them very usable in most PC applications.

Two main methods are used for video compression:

- **Intraframe**, also called spatial compression, in which the data within each frame is compressed individually, with no reference to other frames. Examples of codecs that mainly use intraframe compression are run-length encoding (RLE) and JPEG compression.
- **Interframe**, also called temporal compression, in which the compression concentrates on just the parts of the image that change from one frame to the next. Sometimes key frames are included to reset image data once the changes become excessive. Examples of codecs that mainly use interframe compression are frame-differencing and MPEG.

It is possible to combine these two methods in the same codec.

Figure 12-2 shows a typical second-generation camera. Some signal processing, such as the compression algorithm, is implemented in hardware at the camera, but much signal processing, such as white balance, color correction, dead-pixel correction, and lens correction, are implemented in software on the PC host. Sharing the data processing allows the hardware cost to be reduced while still delivering an excellent product.

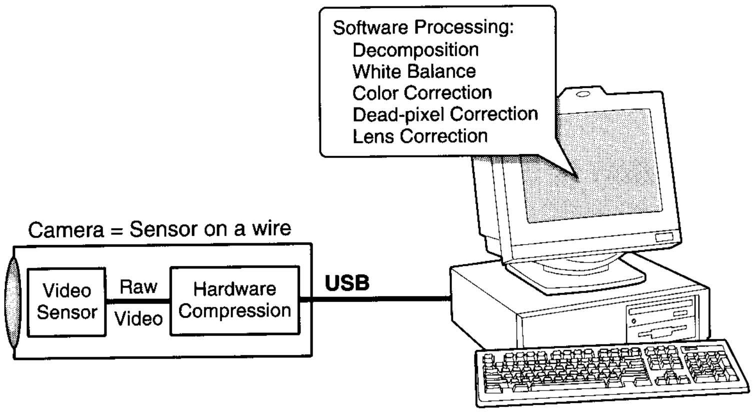


Figure12-2. Digital signal processing is shared

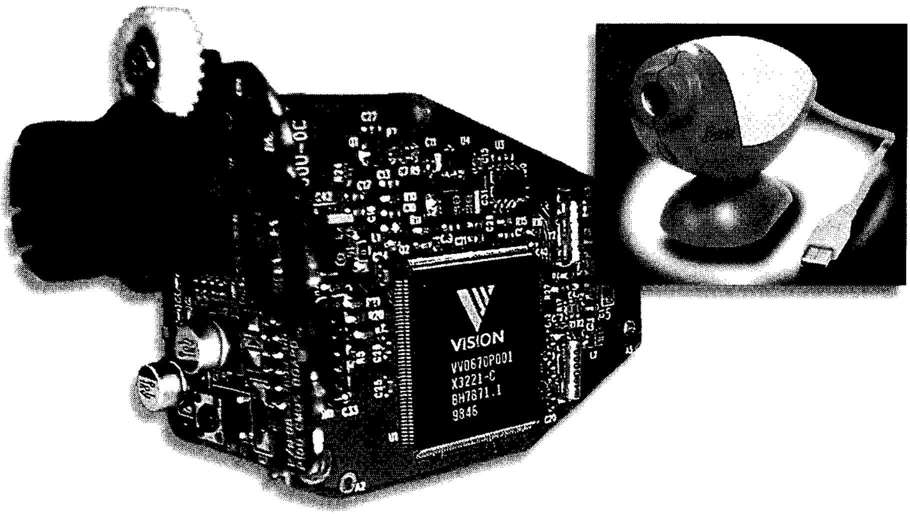
RANGE OF VIDEO SOLUTIONS

Several manufacturers (see Appendix A on the CD-ROM for a full list) have implemented full video-to-USB subsystems. We'll look at a range of example designs and then discuss some new applications that this low-cost, high-quality video is enabling. All of the examples use a custom USB controller with a dedicated task of video capture.

- USB enables a **videoconferencing camera** to be just plugged in to a modern PC—we will look inside a popular USB camera, from Ezonics, and discover how they can ship a product of this quality for less than $100.
- We'll then look at a video-capture example that accepts **composite video** from a camcorder or other NTSC (or PAL or SECAM) source.
- We'll then look at a full **digital video creation** and editing subsystem that can be used to create high-quality movies for a small fraction of the cost of a professional studio.
- Finally we'll look at a range of **USB-enabled video applications** that use digital video as an element of their overall design.

Example 1: Videoconferencing Camera

Figure 12-3 shows the end-user view, and an OEM view, of a videoconferencing camera from Ezonics. The Ezonics product is a turnkey videoconferencing solution, and, because of its high functionality and low cost, the camera will be employed in multiple other applications. Inside the case is a robust video design from VLSI Vision Ltd.; Figure 12-4 shows a block diagram of this OEM solution.



Courtesy of VLSI Vision Ltd. (OEM design) and Ezonics Corp. (consumer design).

Figure 12-3. Video conferencing design

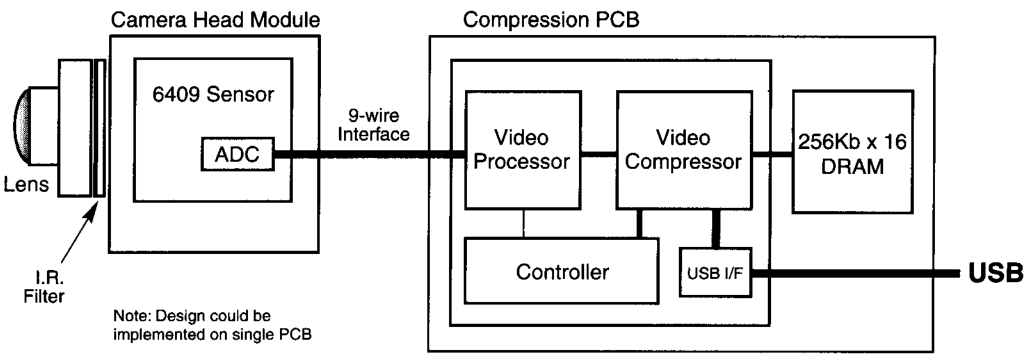


Figure 12-4. VLSI Vision OEM reference design

The CMOS image sensor in the VLSI Vision reference design is a single-chip device that includes the imaging array and all the control logic required to generate digital video. The block diagram (Figure 12-5) includes the photodiode array that is controlled by shift registers. The array generates pixel data that is digitized and formatted into two nibbles per pixel. The sensor also contains control functions for black-level calibration, exposure, image format, and a serial interface to allow control of the sensor from the coprocessor chip.

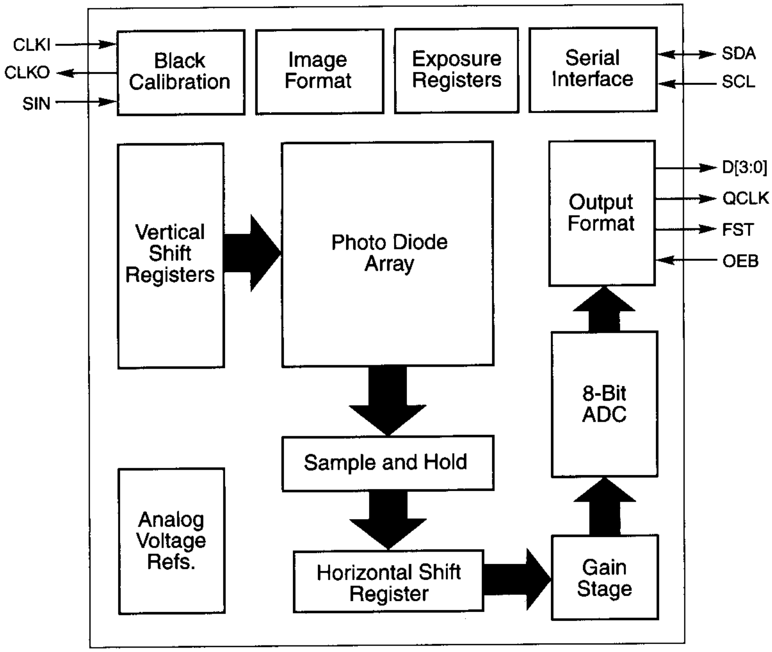


Figure 12-5. CMOS sensor block diagram

The CMOS sensor is controlled by the Color Processor Interface ASIC via a serial bus. Nibble data from the sensor is strobed into the CPIA for processing. The only other component required is a 4-Mbit DRAM, which provides the frame store and run-length buffer. The CPiA performs the color processing, implements a proprietary interframe compression encoder, and manages the USB interface. The whole design takes less than 100 mA, so it can be powered directly from the USB cable.

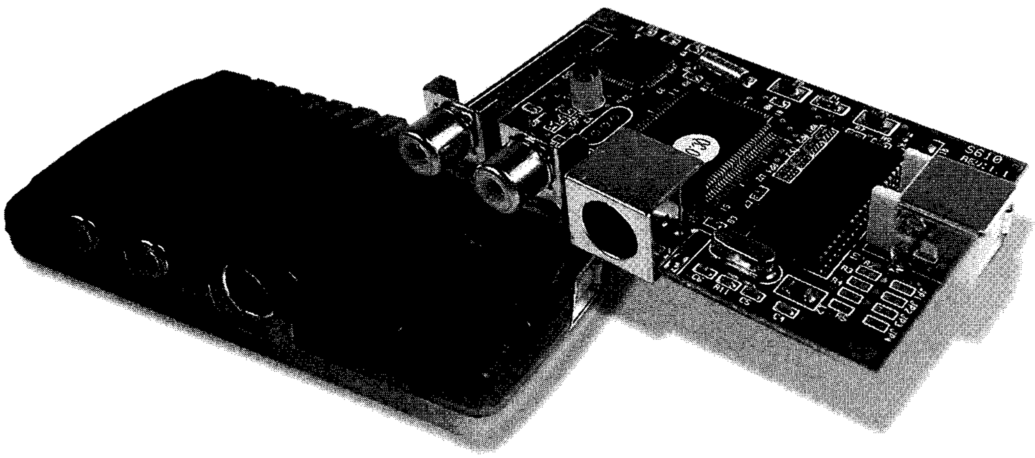
The camera defines a single device descriptor, a single configuration descriptor, and four interface descriptors paired with four isochronous endpoint descriptors. The Default interface requests no USB bus bandwidth as required by the USB specification—this means that the device will be successfully enumerated but

will not use bus bandwidth until an application program selects an alternate interface. The alternate interfaces request more and more of the USB bandwidth.

The highest selection is equivalent to 968 BTF, and this allows a CIF format video to be displayed at 25 fps (depending on the amount of movement in the scene and performance of the PC). If this bandwidth cannot be supported, then an alternate setting 2 requests 712 BTF. A third alternate setting requests 456 BTF, and if this low setting cannot be supported, then the user is requested to remove an existing isochronous device from the USB bus so the new camera can be used.

Example 2: Composite Video

My second example accepts a standard composite video signal (NTSC, PAL, or SECAM), digitizes it, compresses it, and transmits it over USB using only three major components. The reference design for the Nogatech NT1003 is a small video dongle as shown in Figure 12-6. Also shown is a commercially available product from Dazzle Multimedia, the Digital Photo Maker, that implements the Nogatech reference design and adds comprehensive digital image processing software to enable creative users to be immediately productive.



Courtesy of Dazzle Multimedia, Inc. (consumer design, left) and Nogatech, Inc. (OEM design, right).

Figure 12-6. Converting composite video to USB digital video

Figure 12-7 shows a block diagram of the Nogatech Reference Design. The heart of the design is a single-chip hardware video compressor with integrated USB interface. A 4-Mb DRAM is required for frame storage and to buffer the isochronous USB data. A Philips SAA7111A is used to convert the composite video into digitally encoded video for the NT1003. An I²C interface and HID driver are also included to manage the video source, if required, and for system setup.

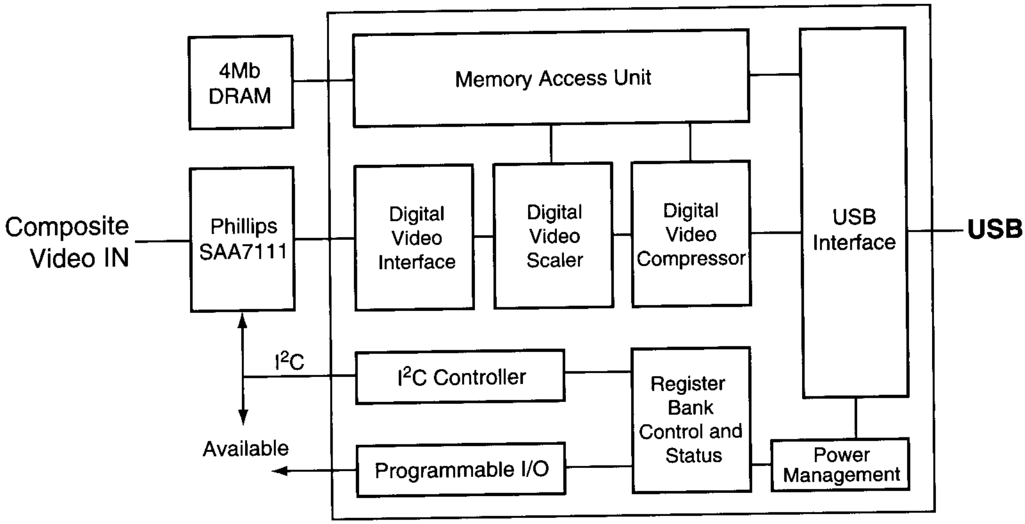


Figure 12-7. Block diagram of Nogatech's reference design

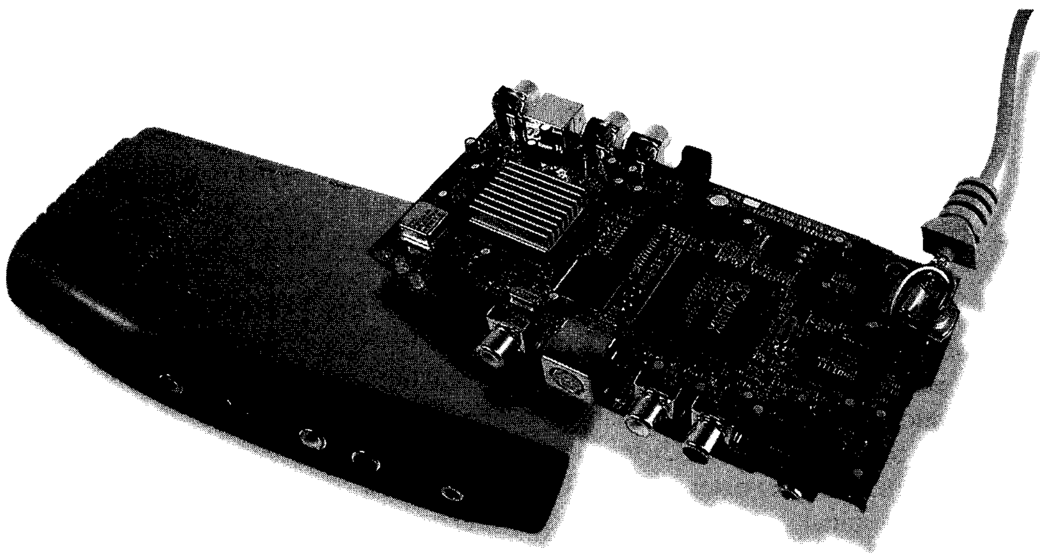
In standard operation the NT1003 will provide the system with the following services:

Compressed Video Channel: The NT1003-1 connects to YUV video digital source, scales the image on the fly horizontally and vertically, compresses the data, and sends it to the host computer via the USB port. The bandwidth usage can be set from 0.5 Mbps to 8 Mbps in 0.5-Mbps increments. The NT1003-1 scaler supports zoom-in–like effects by applying combinations of zooming and cropping built-in functions. The unique method of compression is a special design of Nogatech, to allow easy and fast software-only decompression. The decompression software driver will accept the compressed data and convert it back to standard video formats in less time than it takes to read raw video from any external port.

Video Source Control: Control and status monitoring is implemented with a built-in serial interface, or direct I/O pins. They support a direct-attach camera including a CCD sensor. These controls can also be used by the application software to control and monitor other remote devices. In a videoconference application, for example, this allows the local or remote user to set the focus, zoom, and other parameters of the camera, or even to switch the camera to the power-down mode. The NT1003-1 also supports the use of an external capture button that can be mounted on the camera board and used for capturing still frames on the host computer disk. Still images are captured and sent via the USB in the best quality and resolution that the camera can provide.

Example 3: Digital Video Creation

My third example is a full-digital video creation and editing system from Dazzle Multimedia (Figure 12-8). Figure 12-9 shows a block diagram of this system. The central element is a hardware MPEG compressor from C-cube that Dazzle has integrated with an audio codec and a USB subsystem.



Courtesy of Dazzle Multimedia, Inc.

Figure 12-8. Digital video creation system

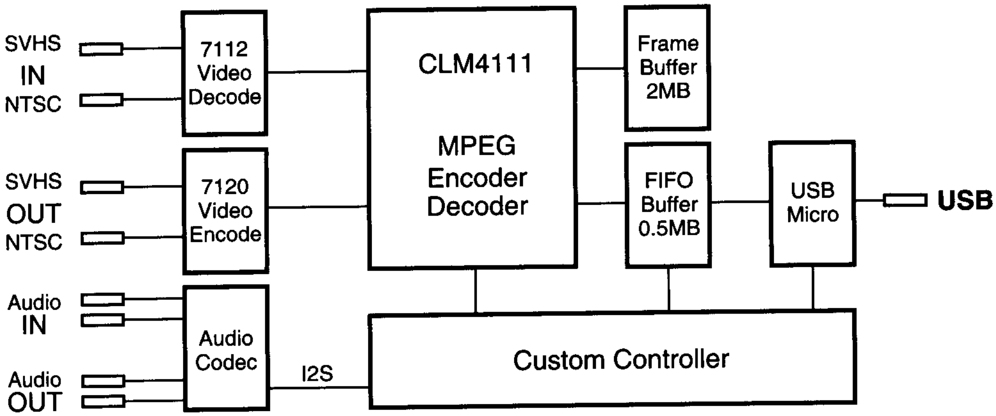


Figure 12-9. Block diagram of the hardware subsystem

MPEG, which stands for Moving Picture Experts Group, defines a set of standards used for coding audiovisual information (such as movies and music) in a digital compressed format.

The major advantage of MPEG compared to other video and audio coding formats is that MPEG files are much smaller for the same quality. This is because MPEG uses very sophisticated compression techniques. The MPEG strategy is to predict motion from frame to frame in the temporal direction and then to use discrete cosine transforms (DCTs) to organize the redundancy in the spatial directions. The DCTs are calculated on 8x8 blocks, and the motion prediction is done in the luminance channel on 16x16 blocks. In other words, given the 16x16 block in the current frame that is to be coded, a close match to that block is sought in a previous or future frame. The DCT coefficients of either the actual data, or the difference between this block and the close match, are quantized, which means that you divide them by some value to drop bits off the bottom end. We hope that many of the coefficients will then end up being zero. The results, which include the DCT coefficients, the motion vectors, and the quantization parameters, are Huffman-coded using fixed tables. This algorithm is well suited for a hardware encoding implementation, and a Pentium processor with MMX instructions can decode this MPEG data to produce a full-screen display in real time.

One side effect of the MPEG encoding is that data cannot be lost—the Pentium processor requires ALL of the encoded data to be able to decode the picture data. So, in contrast with the other digital video solutions presented in this chapter, this MPEG solution uses **bulk** transfers on USB because their delivery is guaranteed. During video capture the USB bus and the PC host's processor will be busy most of the time, and other activity is not recommended because this will result in some loss of video frames.

The USB hardware operates in two modes: It is either inputting digital video and audio into the PC host, or it is outputting digital video and audio from the PC host.

- Let's discuss the input function first. The analog video is digitized by a Philips SAA7112; this multistandard component can create YUV digital video from an NTSC, PAL, or SECAM signal source. This video is compressed at a real-time frame rate by the C-cube CLM4111 MPEG compressor. The compressor needs 2 MB of memory for a frame buffer and for run-length encoding. Compressed output is fed into a 0.5-MB FIFO that is emptied by a USB microcontroller using bulk transfers. At the same time, two audio channels are digitized using a codec, and this data is also sent to the PC host using bulk transfers.
- For playback the PC host will use bulk transfers to move video and audio data to the Dazzle hardware subsystem. The custom controller feeds the video data back through the CLM4111 MPEG encoder-decoder, which uses a Philips SAA7121 video encoder to create NTSC (or PAL/SECAM) video OUT. The custom controller uses an external audio codec to regenerate the sound.

An impressive suite of software enables the real-time capture of video and synchronized audio for later editing. The power of this video studio rivals the capabilities of $3000–$5000 professional installations at one-tenth of the cost.

USB-ENABLED VIDEO APPLICATIONS

Digital Microscope

Figure 12-10 shows a variation on the videoconferencing camera. Intel and Mattel are collaborating on the design of a digital microscope that will be used in schools and will be very popular in the home.

Rather than each student having to peer down an eyepiece to observe a specimen, a USB camera displays an image on the PC screen so that the whole class can see. The image is "live," so that focusing and positional adjustments can be made, and then a higher-resolution image can be captured and stored on disk. These images can be included later in a school report. Science was never this much fun when I went to school!

At home, children will be able to explore their world and then have creative fun doing things with their discoveries that they can show to their friends.



Courtesy of Intel Corp. and Mattel, Inc.

Figure 12-10. A digital microscope from Mattel-Intel

Biometrics

The decreasing cost of capturing complex data types such as images is bringing biometrics applications to the USB-enabled PC. Biometrics is a technology that verifies a person's identity by measuring a unique-to-the-individual biological trait. Biometric technologies include retinal/iris scanning, face-shape recognition, dynamic signature verification, voice recognition, and fingerprint identification.

Biometric identification is superior to lower-technology identification methods in common use today. Today, physical objects or behaviors-based-on-memory are used to identify a user. Physical objects include smart cards or magnetic-stripe cards; behaviors-based-on-memory include the act of entering a PIN number or a secret password.

The primary use of a physical object or behaviors-based-on-memory has a clear set of problems and limitations. Objects are often lost or stolen, and a behavior-based-on-memory is easily forgotten. Also, the use of a valid password on a computer network does not mean that an identity is genuine. These limitations decrease trust and increase the possibility of fraud. They are at the root of widespread distrust of the Internet and are the biggest weakness in true network security. Trust is the problem—without biometrics it is simply not possible to trust or prove that a person is really who they claim to be. Even worse—without biometrics the risk of fraud is very high, while paradoxically the **proving** of fraud is impossible.

Biometrics is a better alternative because it increases trust by creating a context of confidence and undeniable personal responsibility. It is very difficult to share your face or fingerprint. It is nearly impossible to replicate a voice pattern or duplicate the creation of a handwritten signature.

In the past—because of a lack of desktop computer speed and narrow available network bandwidth—there was no practical alternative to the use of physical objects and behaviors-based-on-memory. These old technologies flourished because there was no attainable practical alternative. The future destiny of computerized network security and identification is biometrics. The limiting factors of speed and bandwidth are now a thing of the past. The sensors and electronics to implement biometrics used to be what you saw in science fiction movies—now you can buy them for less than a hundred dollars.

Two interesting biometrics applications are the U.are.U fingerprint analyzer shown in Figure 12-11 and the IriScan application shown in Figure 12-12.

Fingerprint Identification

Image sensors, as described earlier in this chapter, are only the first step in capturing and verifying a fingerprint. The fingerprint sensor is often dirty because of its frequent use, and patterns in the shape of previous fingerprints are already on the sensor before we start. When a new finger is applied, the detector must create a new image knowing the previous dirt on the sensor.

The sensor could send the raw, new image up to the PC host for processing, but this is a security risk! It is easy to observe all of the data transactions on a USB cable, so a clever spy could record the data stream of a valid user's fingerprint and inject this data stream later. To prevent this type of fraud, the U.are.U sensor uses a high-performance USB microcontroller inside the unit to preprocess the data (Figure 12-11). It then manages a secure handshake with PC host-based software and sends encrypted results on the USB cable. Recording the data stream for later misuse will not work!



Courtesy of DigitalPersona, Inc.

Figure 12-11. Fingerprint identification on the PC

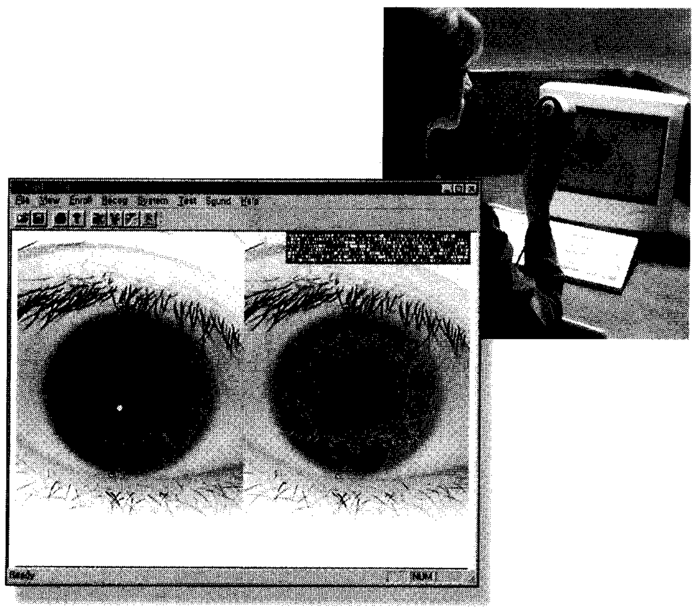
IriScan Application

IriScan's iris recognition technology identifies people by the unique patterns in the iris of the human eye. This capability is possible because no two irises are alike in their structural detail, not even between identical twins or triplets.

A high-definition (640x480x24-bit) image is required for effective iris pattern recognition (Figure 12-12). The intricate and personally distinct patterns of the iris are shaped from a complex information-rich meshwork of tissue. From this special pattern, a unique "bar code" is mathematically computed that is unlike any other in the population. In fact, the identification accuracy of IriScan's iris recognition technology outperforms DNA testing. This pattern, a very personal ID code, is safely and noninvasively acquired by a camera, analyzed, and encoded into a 512-byte IrisCode record. The IrisCode record is used to confirm a person's identity.

The sensor is able to detect eye movement in a live picture, so attempted fraud by using a photograph of a valid user will not work. Sorry! As the world's criminals get more sophisticated, then so must our protection schemes.

In the product configuration shown in Figure 12-12, processing is done on a PC host, requiring a secured cable connection from the imager to the PC host.



Courtesy of IriScan, Inc.

Figure 12-12. IriScan application on the PC

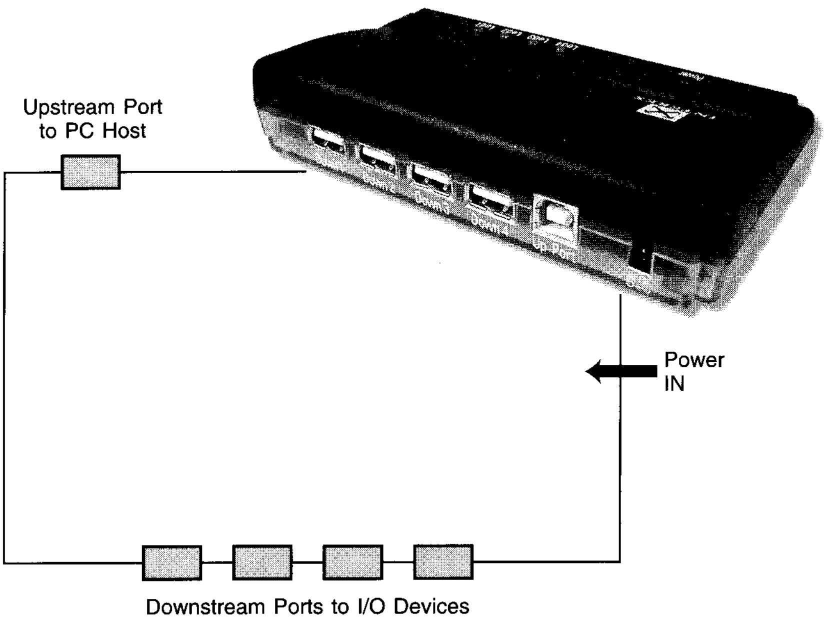
Chapter Summary

Video capture with a PC host used to be expensive and difficult. USB has removed almost all of the complexity and cost from this solution. It is now easy to include video in your custom I/O application. Several manufacturers build custom USB microcontrollers that are specifically designed to compress video so it can be transported efficiently over a USB cable. Some vendors use a proprietary compression algorithm and supply a Windows driver to decompress the digital video. One vendor uses a standard MPEG encoding scheme to implement a combined video and audio solution. This hardware proliferation will encourage software developers to write more creative applications, which will mean more hardware vendors will make even better-quality video devices at lower cost. And this spiral will continue with the consumer gaining the most benefit.

Chapter 13
Designing a Hub

The hub is an essential part of the USB architecture, and its operation contributes significantly to USB's ease-of-use. The hub has two major roles—to provide more connectivity for USB devices, and to provide and manage power for these devices. In this chapter, I will describe the elements of a basic hub and their operation. The USB specification defines a basic hub exactly; in fact, several manufacturers supply a complete, turnkey solution in silicon. Then we'll look at a hub design with a variety of embedded I/O devices.

Figure 13-1 shows a representative basic hub product and its block diagram. It appears to be a simple signal splitter/combiner, but this section will reveal what really goes on "under the hood."



Courtesy of Interex, Inc.

Figure 13-1. A hub provides connectivity and system power

More interesting, from our I/O device perspective, is a compound device that is a hub with embedded I/O devices (Figure 13-2). A compound device can also be thought of as an I/O device with more downstream USB sockets that support the daisy-chaining of more I/O devices.

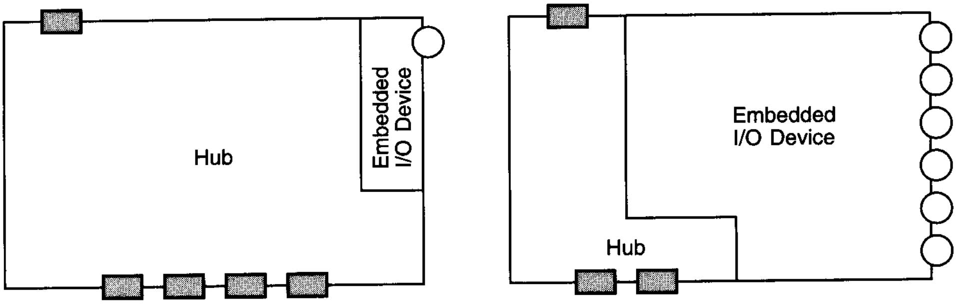


Figure 13-2. Two equivalent views of a compound device

THE BASIC HUB

We can identify three major hub elements: the hub repeater, the hub controller, and the port power control (Figure 13-3).

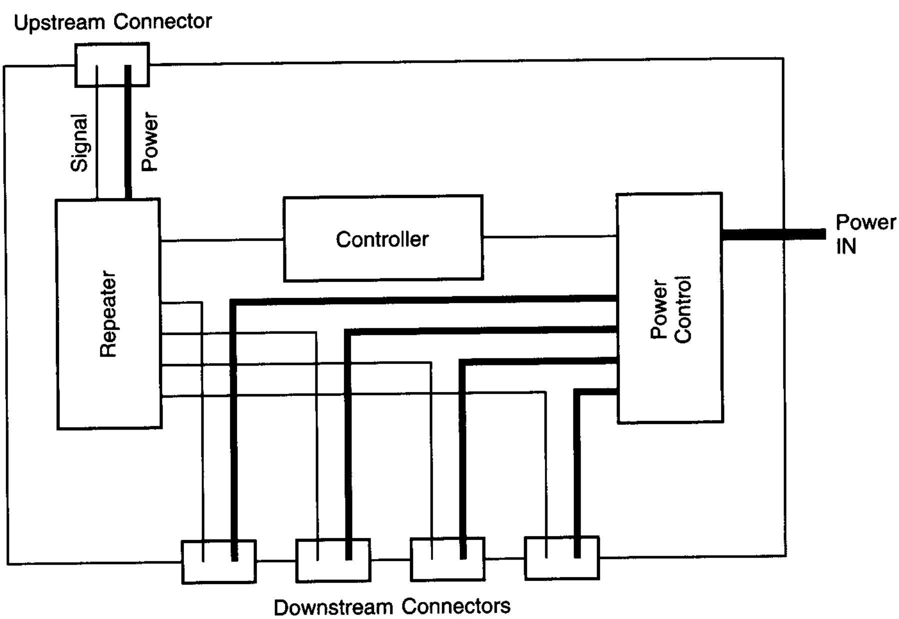


Figure 13-3. Identifying the major elements of a hub

Hub Repeater

The upstream port always operates at 12 Mbps while the downstream ports can operate at 12 Mbps or 1.5 Mbps, depending on the capabilities of the device attached to the port. If no device is attached to a downstream port, it is disabled. To better understand hub operation, let's look at how signals pass through the hub. Let us assume, for the following discussion, that there are full-speed devices on ports 1 and 3, a low-speed device on port 2, and no device on port 4.

The hub decides on a packet-by-packet basis how to propagate signals between the upstream and downstream ports. All packets arriving from the upstream port will be full-speed, and the repeater replicates these packets on all enabled, full-speed downstream ports. Some of these packets will initiate response packets from a device, and in the case of a full-speed device, a packet received on any of the downstream ports is replicated to the upstream port. The USB protocol allows only one "talker"—this is typically the PC host but will be the addressed slave device during the response phase. Therefore, the hub does not have to deal with multiple responses from many downstream ports because this is not permitted.

For full-speed packets, the hub broadcasts downstream packets to all enabled downstream ports and routes response packets from one of the downstream ports to the upstream port. This is summarized in Figure 13-4.

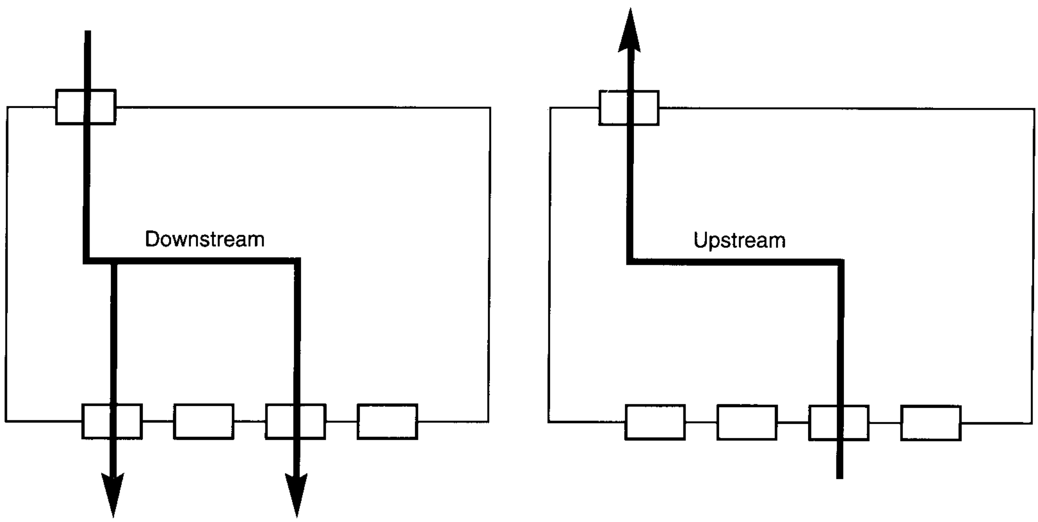


Figure 13-4. Downstream packets are broadcast, upstream packets are routed

Low-speed I/O devices get special treatment. Packets destined for low-speed I/O devices are sent from the PC host with a preamble token, which alerts the hub that the following packet is low-speed. The hub will replicate the preamble token and the subsequent low-speed packet on all enabled full-speed downstream ports and will propagate the low-speed packet, without the preamble, onto enabled low-speed ports. Only low-speed packets are propagated on low-speed downstream ports because of the filtering action of the hub. The PC host will always send at least one packet to all low-speed downstream ports every frame, to prevent them from suspending when there is no other low-speed bus traffic.

A low-speed I/O device will, of course, respond at low speed. The hub replicates this response on the upstream port.

The operation of the hub repeater is autonomous and does not need the assistance of the hub controller. The hub controller's role is to manage system configuration changes.

Hub Controller

A hub is an I/O device. When first attached to a PC host, it is enumerated like any other USB device. Figure 13-5 shows the descriptors that a basic hub will present to the PC host during enumeration. The host software discovers that this is a hub by the Class code (byte at offset 5). A basic hub has one configuration and one interface, and it requires an interrupt endpoint 1 to report status changes. The hub class of devices includes a HUB descriptor.

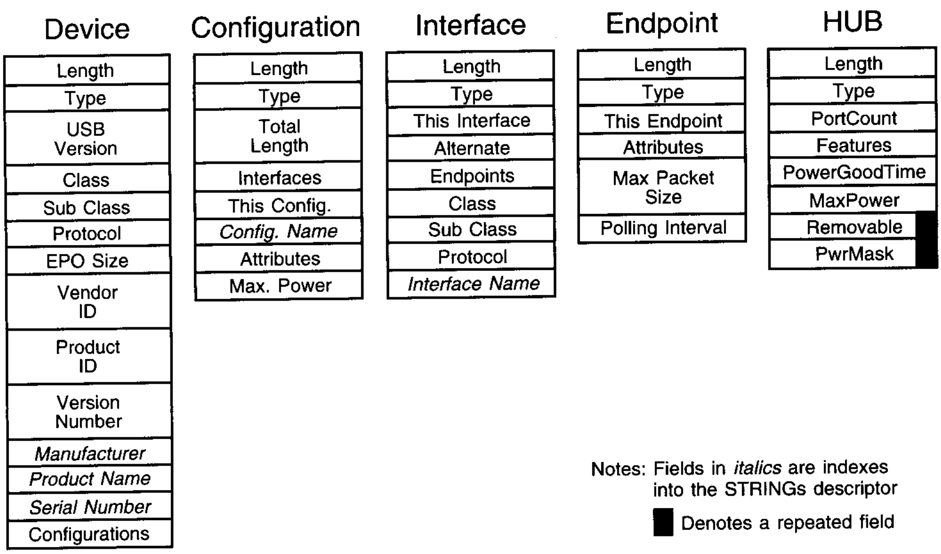


Figure 13-5. Descriptors for a basic hub

We have seen device, configuration, interface, and endpoint descriptors before (Chapter 3), and a hub uses the same definitions. The new HUB descriptor has the following entries.

A hub descriptor has a variable **Length** depending on the number of downstream ports that it is supporting.

The **Type** field is always 9.

The **PortCount** field identifies the number of downstream ports that this hub supports. This will include embedded I/O functions.

The **Features** field is bit-mapped as shown in Figure 13-6.

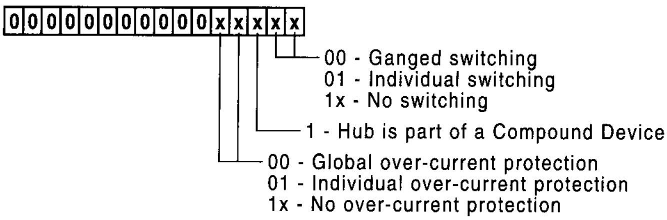


Figure 13-6. Definition of Features bit fields

The time that software should wait after power is applied to a hub is entered in the **PwrGoodTime** field in 2-ms increments.

The **MaxPower** field indicates the maximum current that the hub electronics will consume.

Removable is a bit field that indicates if a device on this port can be removed. Later in this chapter you'll find that this field is important when we add embedded I/O devices to the basic hub. Embedded devices are not removable, and the system software can make some operational optimizations.

PwrMask is also a bit field that indicates if the port has independently switchable power.

As a hub-class device, the hub controller must accept hub-class requests in addition to standard requests. For the full list of requests that must be serviced, see Table 13-1.

Table 13-1. Requests that a hub-class device receives

Type	Request	Our Action
Standard	Get_Status	Return current status
Standard	Clear_Feature	Clear Specified Feature
Standard	Set_Feature	Set Specified Feature
Standard	Set_Address	Store Unique USB Address and use from now on
Standard	Get_Descriptor	Return requested descriptor
Standard	Set_Descriptor	Optional: Set Specified Descriptor
Standard	Get_Configuration	Return Current Configuration or 0 if not configured
Standard	Set_Configuration	Set Configuration to the one specified
Standard	Get_Interface	Optional: Return Current Interface
Standard	Set_Interface	Optional: Set Interface to the one specified
Standard	Sync_Frame	Optional: Synchronize USB Frame Numbers
HUB Class	Get_Bus_State	Optional (for diagnostics): Return D+, D–
HUB Class	Get_Hub_Status	Return Hub Status with Changed Identified
HUB Class	Get_Hub_Descriptor	Return Hub Descriptor
HUB Class	Set_Hub_Descriptor	Optional: Set Hub Descriptor
HUB Class	Set_Hub_Feature	Enable a standard hub feature
HUB Class	Clear_Hub_Feature	Disable a standard hub feature
HUB Class	Get_Port_Status	Return Port Status with Changed Identified
HUB Class	Set_Port_Feature	Enable a standard port feature
HUB Class	Clear_Port_Feature	Disable a standard port feature

Power Control

Port power control consists of overcurrent protection, power-switching, or a combination of the two. Power can be supplied to all downstream ports in parallel or individually. It is cheaper to supply power to all downstream ports in parallel because fewer devices are required, but an overcurrent fault on one port will cause the other ports, in the same hub, to trip also—not a good user experience. Individually powered ports would also include separate overcurrent indicators that can be supplied to the operating software so that specific error messages can be presented to the user.

It is possible to derive the output power for the downstream ports from the upstream bus port. This would be called a bus-powered hub, and I do not recommend this approach. A total of 500 mA is available from a USB port, so a bus-powered hub would need to redistribute this power to its downstream ports. The hub electronics use 100 mA and could provide only 100 mA for each of four downstream ports. If an I/O device that needed more than 100 mA for operation were attached to this bus-powered hub port, it would be enumerated but not enabled. This could confuse many users. I recommend self-powered hubs, which means the hub has a separate power-in connector, so that 500 mA is always available for the downstream ports. A future release of the Windows operating system will alert a user if a high-power device is attached to a low-power hub socket. The Windows software will recommend an alternate attachment point if possible.

The USB Specification defines the maximum amount of voltage drop and minimum acceptable output voltages. For bus-powered hubs, the industry has responded by supplying power switch devices with low series resistance, devices such as the Texas Instruments TPS2014/5 and the Micrel MIC2525 (datasheets are reproduced, with permission, in the Chapter 13 directory on the CD-ROM).

To meet the voltage drop, droop, and EMI requirements in the USB Specification 1.1, careful PCB layout is necessary. The following guidelines must be considered:

Keep all Vbus traces as short as possible and use at least 50-mil, 1-ounce copper for all Vbus traces.

- Avoid vias as much as possible. If vias are necessary, make them as large as feasible.
- Place the output capacitor and ferrite beads as close to the USB connectors as possible.
- Use a separate ground and power planes if possible.

The USB Specification also defines overcurrent protection for the downstream ports. Overcurrent protection is required on self-powered hubs and may be implemented with a resettable PTC such as Raychem's miniSMDC150 or a power distribution switch such as Texas Instruments TPS2014/5 (data sheets and application information are included, with permission, in the Chapter 13 directory). Some hubs, such as the Philips ISP1122, integrate individual power control and the overcurrent protection. Power control is performed using a pMOS transistor on each port, and the "ON" resistance (Rds) of the transistors is used to trigger an overcurrent condition when too much current passes through them. The Philips ISP1122 reference design (included in the Chapter 12/Philips directory on the CD-ROM) details this feature in their Figure 8. Figure 13-7 shows a typical downstream port within a hub.

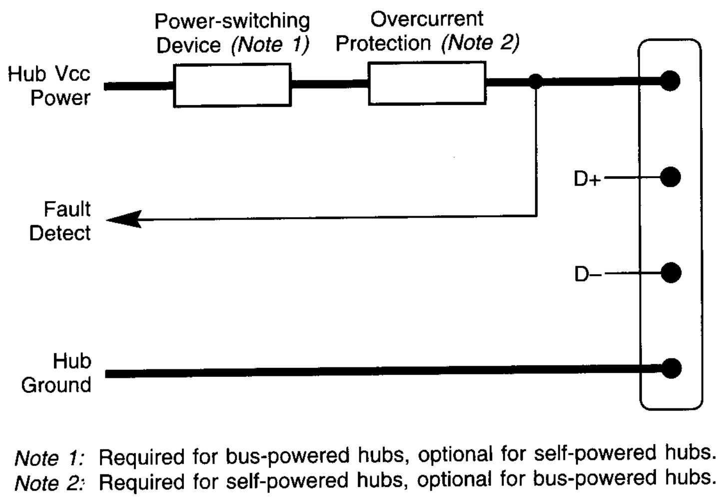


Figure 13-7. Power management of a downstream port

Basic Hub Summary

The operation of a basic hub is well defined by the USB Specification. Many implementations exist (see Appendix A) that are broadly categorized as hard-logic controlled or microprogrammed. Hard-logic devices, such as the Texas Instruments TUSB2043 and the Philips ISP1122, operate as a hub with no extra effort. You just "connect the dots" with careful PCB layout as recommended by their reference design, and you have a working hub.

A microprogrammed device, such as the Intel 8x930HX or the STMicroelectronics ST92161, includes a microcontroller that executes a program to implement the hub controller functions. The hub repeater functions are hard-wired in these devices to meet the timing constraints of the USB Specification. The manufacturers of these devices supply prewritten code for a hub design, so this approach is also connect-the-dots for a working hub.

The design freedom comes in the power controller, but I recommend self-powered with individually controlled outputs.

BUILDING A COMPOUND DEVICE

Now that we understand the construction and operation of a basic hub, let's look in detail at the design of a hub with I/O functionality, or at an I/O device with hub functionality—both are equivalent descriptions of a compound device. Figure 13-8 looks inside a typical compound device.

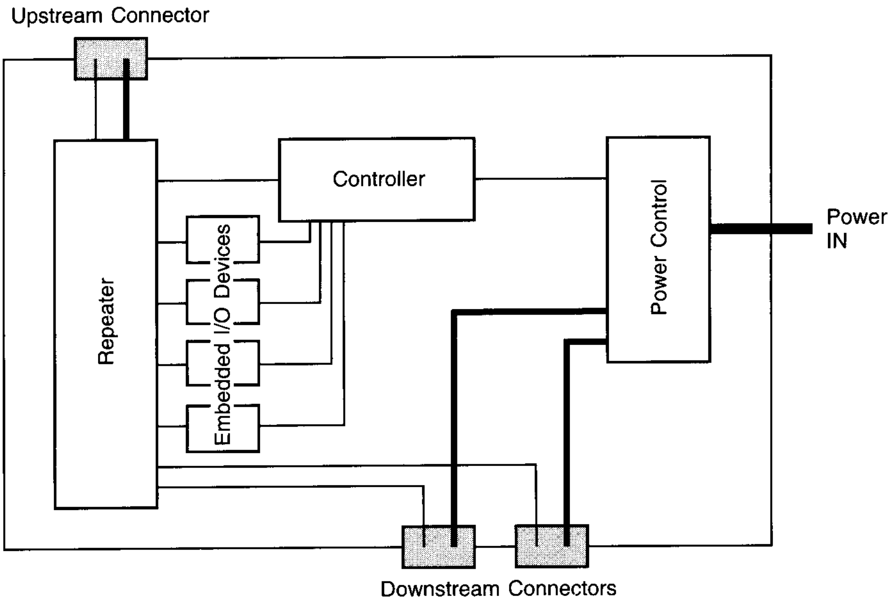


Figure 13-8. A compound device has embedded ports

The number of "external" downstream ports and the number of "internal" downstream ports is ours to decide. An external downstream port is exactly what you would find on a basic hub. Embedded I/O functions are implemented on internal downstream ports. The PC host is unaware of this "external" and "internal" hardware implementation—it just sees devices connected to a hub. A compound device is a hardware optimization that gives the hub controller a little more work to do but saves us the cost of a USB interface and separate controller.

DESIGN EXAMPLE

A convenient place to put a USB hub is inside the monitor of a PC system. An alternative "computer" place is the keyboard, but I prefer the monitor because it has power connected that allows the hub to be self-powered. Many other examples could be chosen. I'll use a monitor example as a vehicle to explain the design but will be as flexible as possible so you can fit the concepts into your application.

To make this example a little more interesting, I am going to design a range of monitors as shown in Figure 13-9. The entry system is just a monitor—it has a VGA connector on the back and a set of buttons that control the adjustment of the monitor picture (size, position, pincushion) in a simple base. The models get more elaborate as I add more features—the first has a USB hub in the base. Then I add a combination of features to produce higher-end products—features such as an IR data transfer port, ports for a keyboard and mouse, speakers, microphone, perhaps even parallel and serial ports and a smart card reader. The flexibility of the solution allows easy configuration of any combination of features.

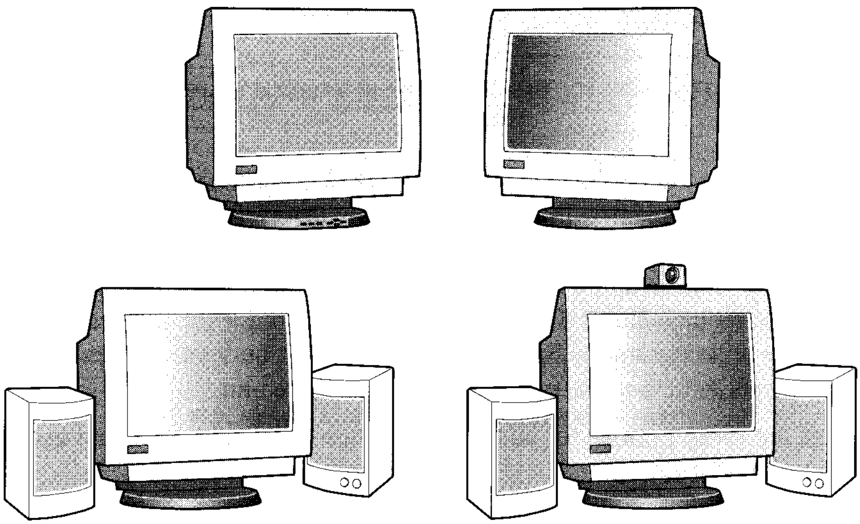


Figure 13-9. A range of monitor solutions

Figure 13-10 shows the internal block diagram of a typical monitor with its connection to the PC. It is microcontroller-based with an I^2C bus interconnecting the building blocks.

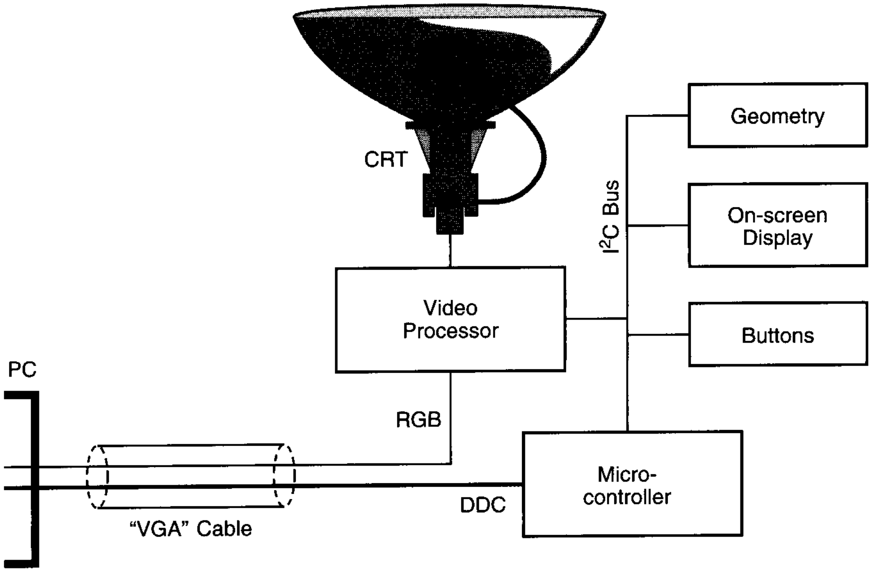


Figure 13-10. Block diagram of a typical monitor

The connection to the PC is minimally a VGA connection. Monitor manufacturers typically implement a Display Data Channel (DDC) using an I^2C bus in the same cable that is used in their manufacturing group for monitor alignment. The lack of a standard Windows driver for this connection has hampered its inclusion on standard video cards, so, for the foreseeable future, monitors will continue to have separate alignment buttons. Our example design implements these buttons in an exchangeable base unit.

There are many options for adding a USB hub to this example, and each involves different tradeoffs.

Step 1: Adding a Hub

If we just want to include a basic hub, then a ready-built hub with I^2C connection, such as a Texas Instruments TUSB2140 or a Philips PDIUSBH11A, would be a good choice (Figure 13-11). These components appear as slave I^2C peripherals to the monitor microcontroller, and we would need to add minimal firmware to the monitor microcontroller to support them.

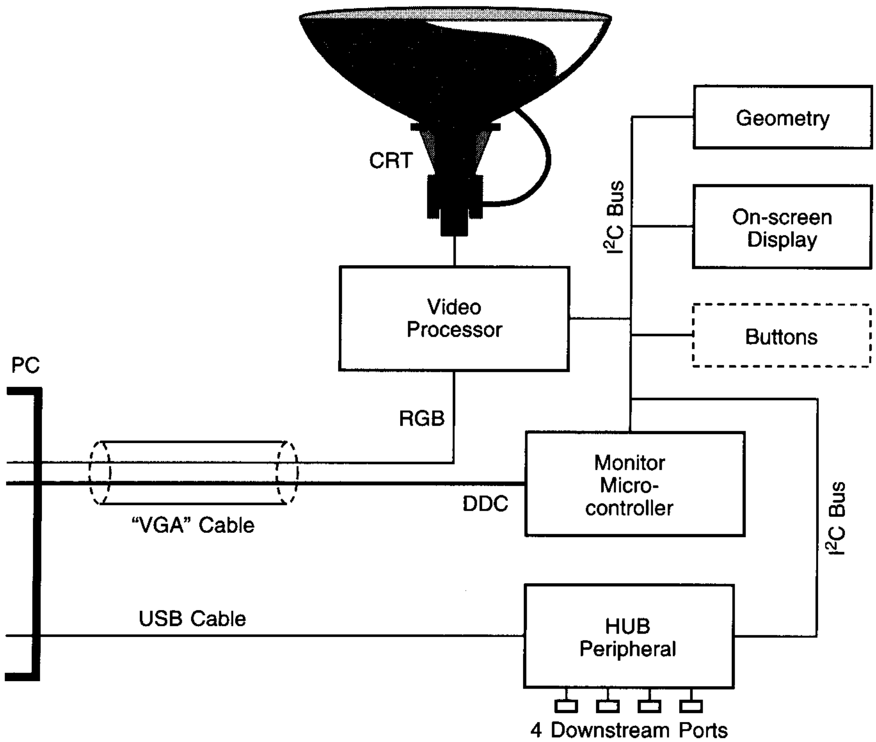


Figure 13-11. Using an I^2C connected hub

The I^2C bus from the USB hub peripheral component is addressable via USB (remember our examples in Chapters 6 and 8). Low-bandwidth data can be transferred between the PC host and the monitor microcontroller using this I^2C embedded I/O function.

We'll replace the physical alignment buttons on the monitor with software on the PC host. The command set required to implement picture adjustment using I^2C has been standardized by a VESA working group (VESA = Video Electronics Standards Group at www.VESA.org), and the document is included in the Chapter 13/VESA directory on the CD-ROM. Figure 13-12 shows a demonstration program written by STMicroelectronics that sends these picture adjustment commands via USB and the hub's I^2C bus to the monitor microcontroller. When the monitor microcontroller powers up, it looks for a "button-base" or a "hub-base" on its I^2C bus and operates accordingly.

We have replaced the cost of the buttons with the cost of the hub circuitry; to put it another way, we got a USB hub for free!

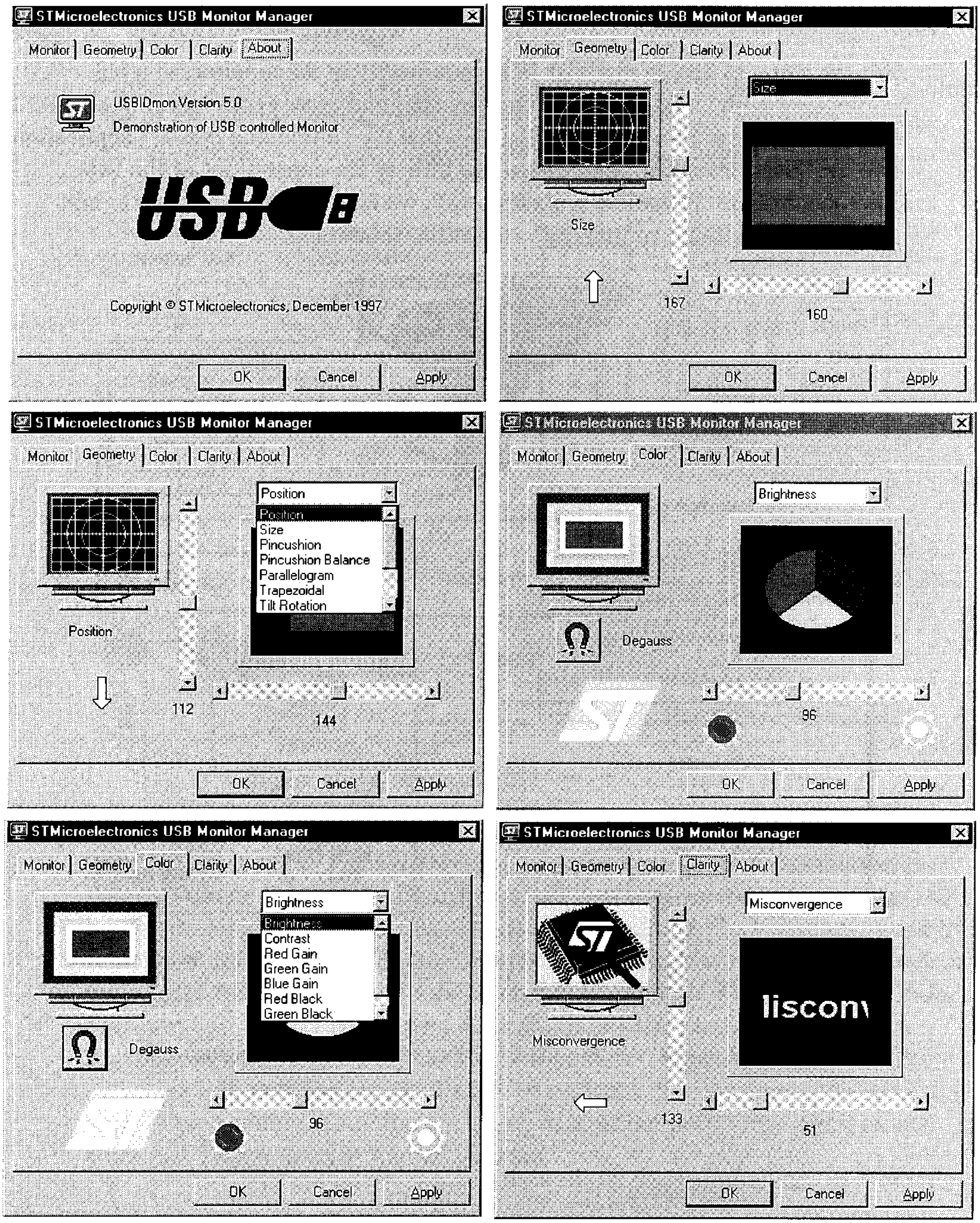


Figure 13-12. An application controls monitor alignment

Step 2: Adding I/O Devices

We are going to design modular I/O functionality in the monitor, so a hub component with an integrated microcontroller and more endpoints (Figure 13-13) would be a better choice. We'll discover that the "alignment button" application and the "hub controller" application use little of the capabilities of the hub microcontroller. To look at it another way, we have ample headroom to add functionality without adding high cost to the solution.

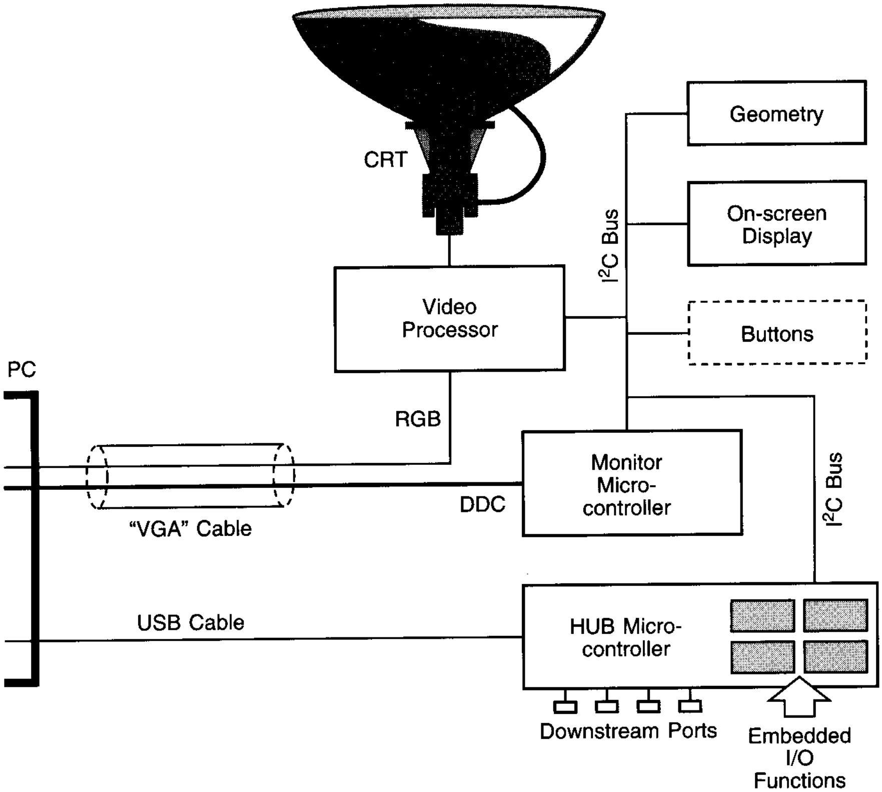


Figure 13-13. Adding I/O flexibility with a microcontroller

Step 3: Extensible Design

Figure 13-14 shows an example of a high-end monitor implementation. The left side of the diagram shows the hardware and the right side shows the software. For full schematics and application software, see the Chapter 13 directory on the CD-ROM.

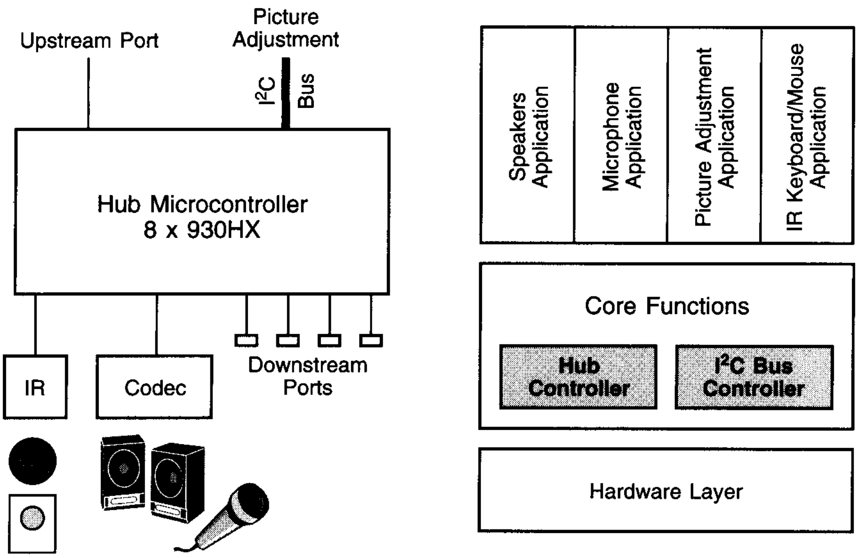
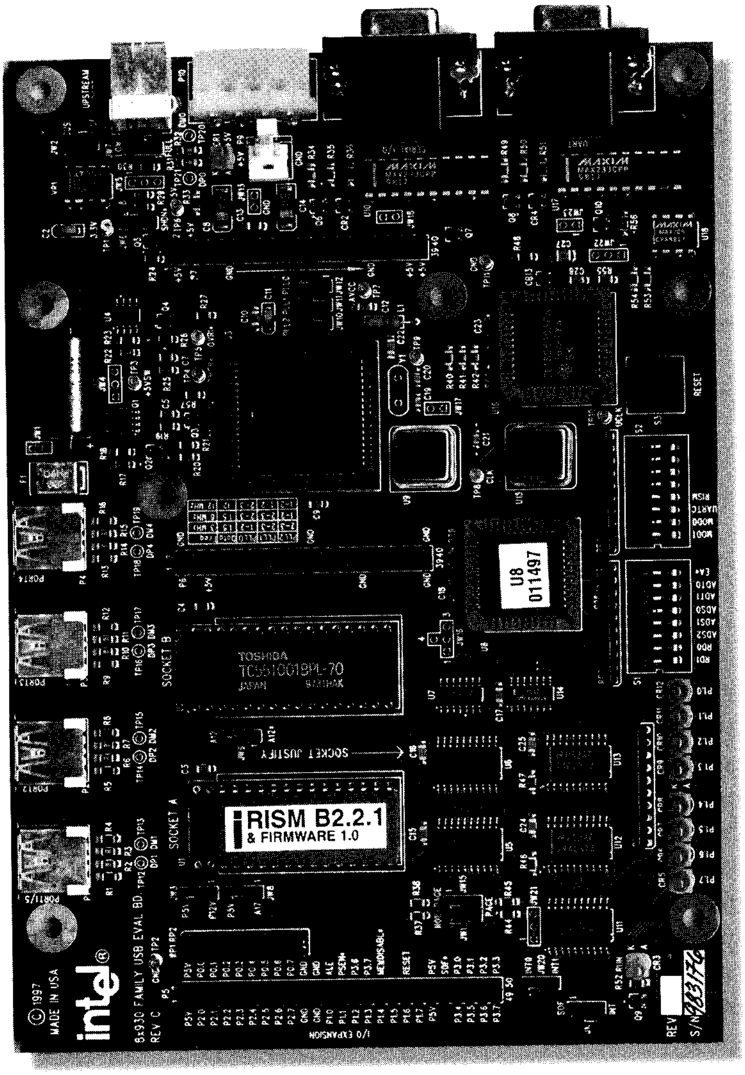


Figure 13-14. High-end USB monitor example

The core firmware to manage the hub and embedded function capabilities is included on the CD-ROM. Descriptors and applications code to match the I/O hardware is added on top of this core code in a building-block fashion. Each I/O function is described by a unique interface descriptor that specifies independent endpoints. All of these interfaces will run concurrently. The example includes a four-port hub, picture adjustment controls via the I^2C bus, speakers and microphone using asynchronous endpoints, and an IR data transfer port. There is still capability with the Intel 8x930HX component (three unused endpoints) that would allow adding more functions such as a printer port, serial ports, mouse, keyboard, floppy disk, or a smart card reader.

A videoconferencing camera needs its own special-purpose controller to implement data compression, so this could not be added as an application on top of the core firmware. A camera could be attached to one of the downstream hub ports (we will do this in the next chapter). All these features and we have only one USB connection back to the PC host! Figure 13-15 shows a photograph of the completed example design.



Courtesy of Intel Corp.

Figure 13-15. Incremental features for a monitor solution

CHAPTER SUMMARY

This chapter has shown that a hub design is very straightforward. Several manufacturers build plug-and-go hub components that are hard-coded to operate in a USB environment. Care must be taken on the power management and overcurrent protection, and detailed design recommendations are included on the CD-ROM. Some hub components use an embedded or external microcontroller to implement more embedded I/O functionality—the firmware to implement the base functionality is often provided by the component manufacturer.

Embedded I/O devices are easy to add to a hub controller that supports multiple device endpoints. The design methodology is the same as that for a stand-alone I/O device. Modules of code to support each interface can even be moved from a stand-alone I/O device into an embedded device supported by the same microcontroller.

I/O devices embedded in a hub give the USB designer an additional degree of freedom when designing a system. And, as we have seen, the design is not difficult.

CHAPTER 14
PUTTING IT ALL TOGETHER

The previous chapters have described buttons and lights (HID devices), audio, video, and hub design as discrete elements. This chapter will show how easy it is to integrate the multiple diverse elements that we have covered in the previous chapters into a single, very functional device. We shall see how USB interface descriptors can be added to an I/O device and how the PC host interacts with each independent interface. The building-block nature of USB will be very evident and will promote the reusability of software and hardware elements. The example I have chosen is a TV-tuner dongle. This bus-powered device is implemented as a collection of devices and a stand-alone hub, packaged in a single enclosure as shown in Figure 14-1. The video will be displayed on the PC monitor, the sound will be heard on the PC speakers, and channel tuning will be implemented via a Visual Basic application on the PC.

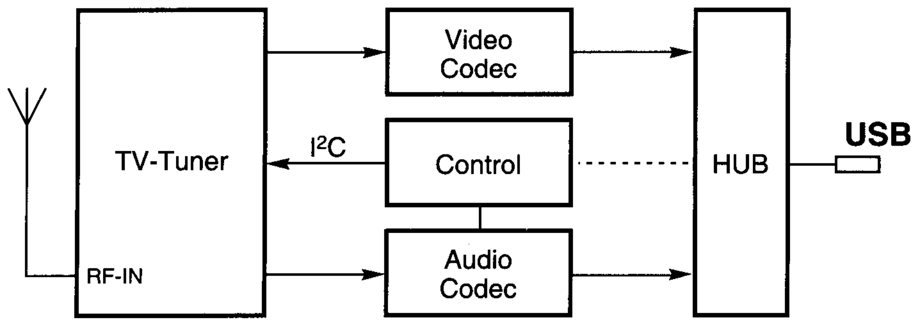


Figure 14-1. Block diagram of TV-tuner dongle

OVERVIEW OF DESIGN EXAMPLE

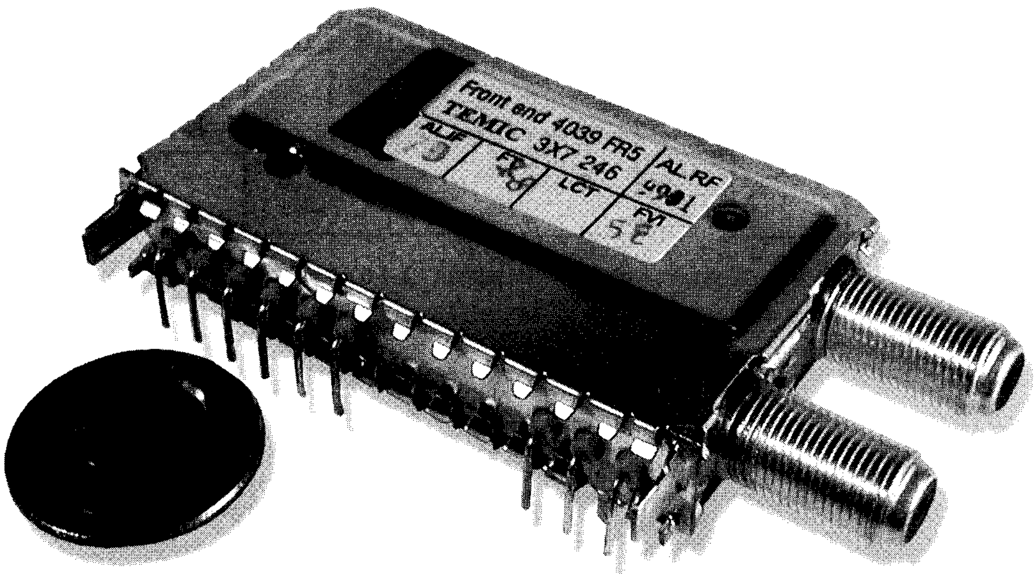
The heart of this example is a prebuilt TV-tuner specially designed for multimedia applications. The tuner provides the analog front-end processing of cable or antenna-based RF signals, to supply a composite video output and stereo audio outputs. All of the setup and channel selection of the tuner is implemented via an I^2C bus. The tuner's composite video output is fed into a video-to-USB subsystem that was presented in Chapter 12. The tuner's audio outputs are fed into a stereo codec subsystem that was presented in Chapter 11. A hub is included inside our dongle to combine the video subsystem and the audio subsystem into a single USB cable, as was presented in Chapter 13. An I^2C control bus must be generated, and we have several implementation options. This full design needs to consume less than 500 mA so that it can be implemented as a bus-powered device.

The design looks like a composite device, but it is implemented as a stand-alone bus-powered hub with two bus-powered devices permanently attached and packaged in a single enclosure. The PC host software will not realize that we have a collection of independent devices in a single enclosure, so we will have no special software to write on the PC host (always a plus!). The PC host will first enumerate the hub; then it will enumerate the audio device and the video device as separate devices, which will cause separate device drivers to be loaded.

The hardware for this example neatly partitions into five subsystems: tuner front end, video-to-USB, audio-to-USB, I^2C control, and hub/power distribution subsystem.

Step 1: Tuner Front-end Subsystem

This design example uses a Temic 4039 FR5 (Figure 14-2). There is a lot of analog magic inside this module that we fortunately do not have to deal with! Some very clever analog designers have integrated a hyperband tuner that covers a frequency range from 54 to 805 MHz (so it can be used worldwide), an IF-strip with SAW-filter, IF-amplifier, and video and sound demodulators into a single tested module that is controllable digitally via an I^2C bus. For the full data sheet, see the Chapter 14/Temic directory on the CD-ROM. I chose the Temic 4039 because it was the smallest physical size for our dongle implementation. The data sheet also describes FM signal reception that is not highlighted in this design example. The 4039 includes complete FM signal processing, including demodulation and stereo decoding. The example design includes all of the hardware to support tuning to FM stations, but a human interface on the PC host is not implemented in this example design—the current source code is provided for an interested reader to add these capabilities.



Courtesy of Temic Telefunken Hochfrequenztechnik GmbH

Figure 14-2. TV tuner from Temic

The 4039 module requires no external components. You hook up +5 volts and it works! Some initialization is required, and this is implemented by the I^2C control subsystem.

Step 2: Video-to-USB Subsystem

The Nogatech video example from Chapter 12 is used. The Nogatech reference design is self-contained and includes all the required descriptors and endpoints to manage the video capture and conversion to USB isochronous packets. The design includes an I^2C control bus that is intended to operate a camera; we are using a tuner instead of a camera in this design example, and we could use the I^2C bus to control the tuner. For the circuit diagram for the video subsystem, see Figure 14-3.

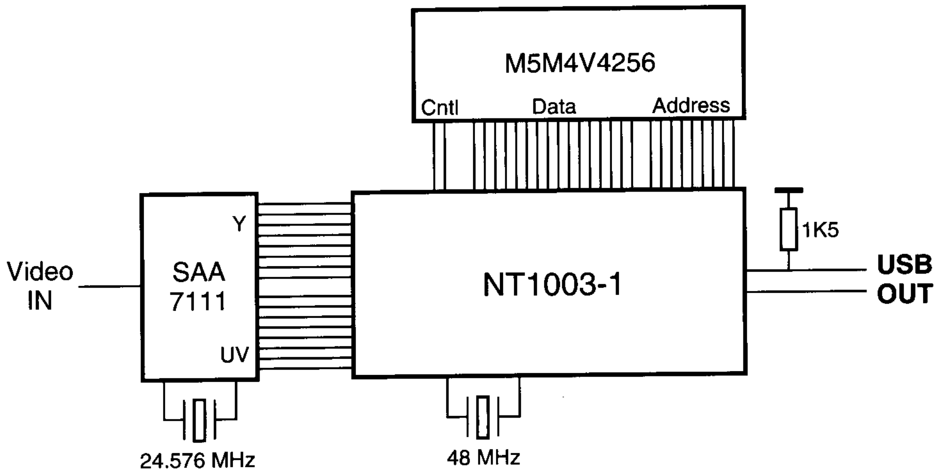


Figure 14-3. Video subsystem schematic

Step 3: Audio-to-USB Subsystem

Chapter 11 discussed the various options of dealing with audio. I chose the Philips UDA1335 for this design example because of its small size and simplicity.

The UDA1335 contains all of the descriptors and endpoints required to fully implement the audio capture task. It also includes software-programmable volume, bass and treble, and an I^2C interface that could be used to control the TV tuner. Figure 14-4 shows the circuit diagram for the audio subsystem. The audio OUT channels of the UDA1335 are not used in this example, although you are free to use them if you choose.

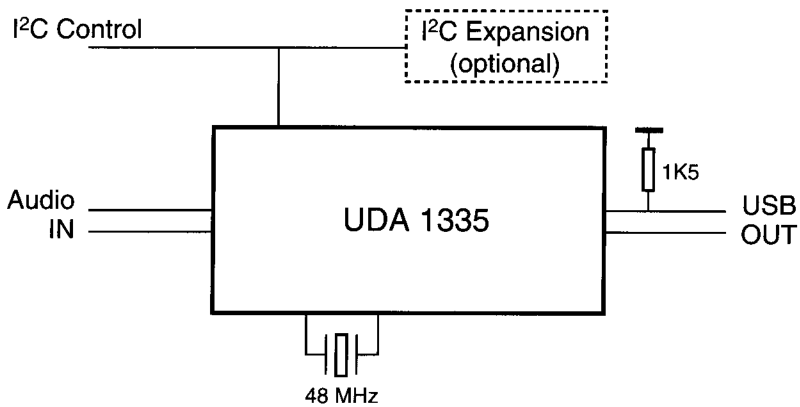


Figure 14-4. Audio subsystem schematic

Step 5: I²C Control Subsystem

We have three choices to implement an I²C control bus:

- Use I²C from the Nogatech video subsystem.
- Use I²C from the Philips audio subsystem.
- Use a microcontroller with an I²C interface on a spare downstream port.

The third option would be the least design effort because we could just copy the example from Chapter 9. This is also the most expensive from a unit-cost perspective. As it turns out (look ahead to step 6), we do not have enough power available for a microcontroller if we want to keep the total dissipation below 500 mA. So we must choose the video or audio I²C solution.

The Nogatech and Philips designs use independent Vendor Defined commands to control the I²C bus. This is a little different from the HID approach in Chapter 9, so we will require a small change to our application software

Step 6: Power Distribution

The components require a mix of 5V and 3.3V as shown in Table 14-1.

Table 14-1. Maximum power consumption

	5V	3.3V
TV tuner	230	—
Video subsystem	—	73
Audio subsystem	4	56
Hub electronics	—	100
Total		**463 mA**

We want the total power dissipation to be less than 500 mA so that we can power the dongle from the bus. A 3.3V voltage regulator will be required for the audio subsystem and the hub electronics. One regulator is already included in the video subsystem design, and this has the capacity to supply up to 400 mA for a camera. Because the design doesn't include a camera, we can use the power for the audio and hub electronics.

Step 7: Design Optimizations

The hardware subsystems were designed independently. As a result, there are more components that a little extra design work could eliminate—for example, there are three 48-MHz crystals. If I were taking this design into production, I would apply more effort to reduce the component count while still maintaining a reliable and robust design.

The video, audio, and hub subsystem could be manufactured on a board approximately the same size as the TV tuner case (3.0 inches x 1.5 inches) as shown in Figure 14-6. When the board is mounted in a plastic case, we have created a highly functional and desirable dongle.

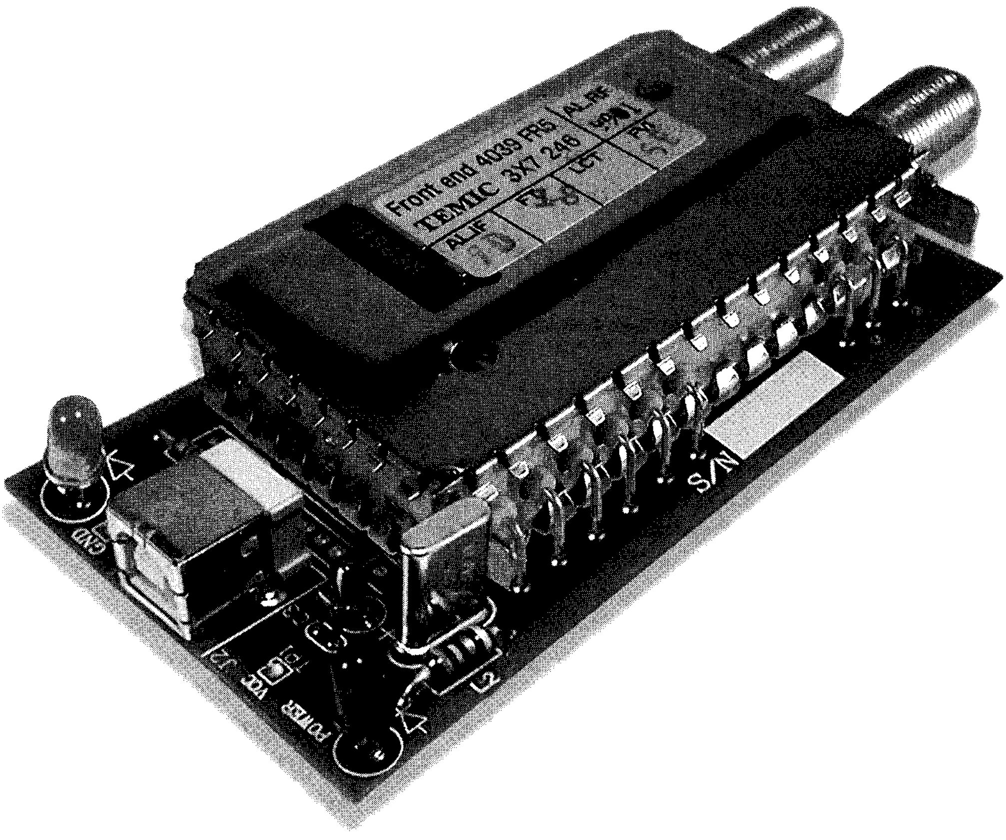


Figure 14-6. Representative dongle design

Step 8: PC Host Application

The Windows operating system includes drivers for the video and audio subsystems, so all that is required for the PC host application is some human interface design and the control task to change bands and channels on the TV tuner. This software stack is shown in Figure 14-7 with the human interface program (the one we have to write) highlighted.

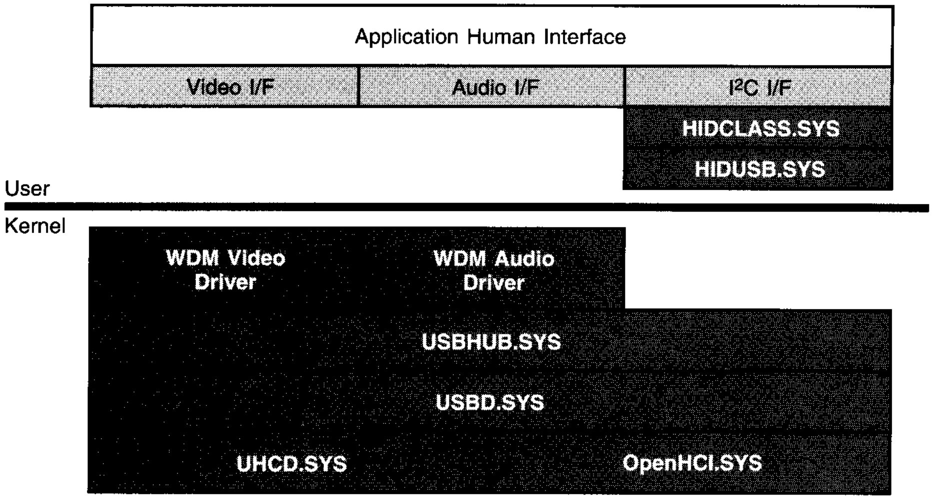


Figure 14-7. Most of the required software is included in the operating system

Figure 14-8 shows a representative human interface that operates with the control panel applet for audio volume, balance, and tone. The program code behind the channel-select buttons is included in the Chapter 14 directory on the CD-ROM. The code basically consists of a look-up table that sends the correct I^2C control codes to the tuner using the program structure that was developed in Chapter 6. Vendor-defined commands are used to control the I^2C bus because that's how the chosen components operate.

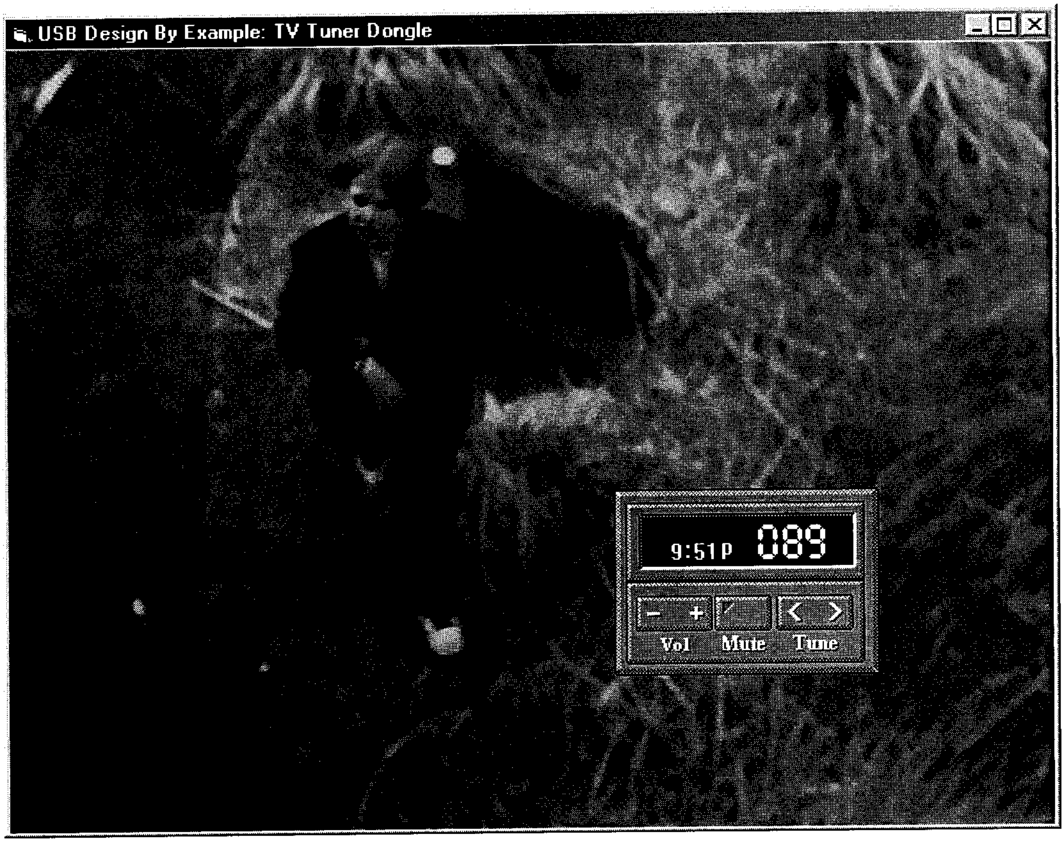


Figure 14-8. Human interface for the TV tuner

CHAPTER SUMMARY

Can you believe it? We have just implemented a TV tuner dongle. This bus-powered device collects TV station input from an antenna or from cable an displays the result on the PC host monitor screen with full stereo sound! A sing USB connection manages video, audio, and channel selection—pretty impressive.

The design was implemented using a building-block approach for both the I/O device hardware and the PC host software. Each building block has an interfac definition: The PC host software interacts with an interface to provide or consume data, and the I/O device accepts or provides data via this interface and manages real-world hardware corresponding to this data. A single USB device can have multiple interfaces, each operating independently, enabling rich applications to be quickly realized. The USB architecture was designed to enab comprehensive applications to be implemented as a set of building-block elements.

CHAPTER 1
INCREASING DATA BANDWIDT
INTO THE HOM

This final chapter takes a forward look at some exciting applications that are being enabled by USB technology. While designed to give you a flavor of the possibilities and expanded scope of products that USB can deliver, many of the examples in this chapter have already been prototyped by Intel, and complete reference designs are available for license. If you would like more information about the projects discussed here, please e-mail the author at john@usb-by-example.com.

PC communications is a rapidly growing application area that is complemented by USB technology. We'll look at three existing media types currently supplyin information into the home (telephone line, TV cable coaxial cable, and satellite broadcast), and we'll investigate how we can better use this pre-existing infrastructure for computer-to-computer communications.

We will discover that USB makes this application area quite straightforward an creates many possibilities that are really exciting.

THE PLAIN OLD TELEPHONE SERVICE (POTS)

The analog telephone line that the telephone company brings up to your home, and the telephone wiring inside your home, is much underused. The bandwidth required to support dialing and speech is only 20 Hz to 3.4 kHz. And the installed wires are capable of supporting 20 Hz to 10 MHz! Figure 15-1 shows a proposed bandwidth allocation of these telephone wires.

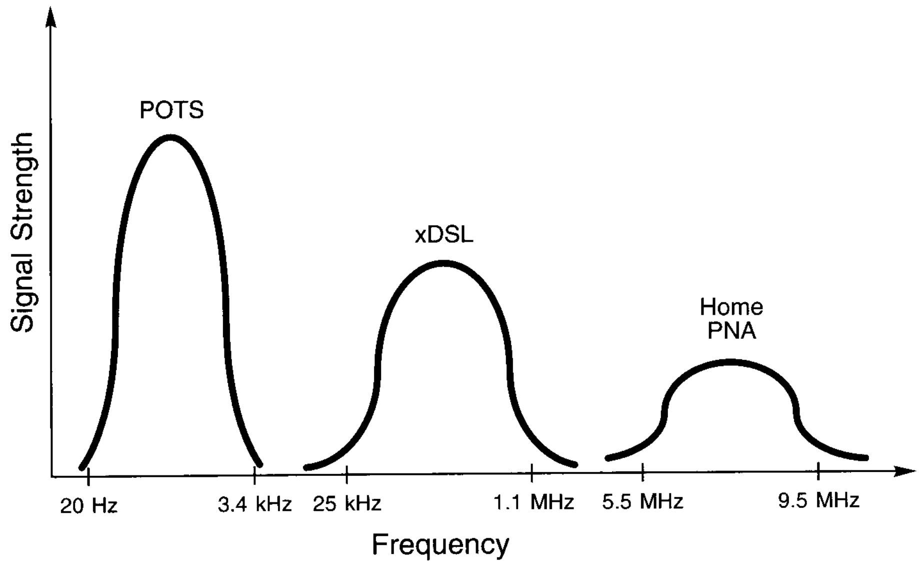


Figure 15-1. Proposed bandwidth allocation of telephone wires

The POTS bandwidth allocation is for a standard analog telephone.

xDSL is a Digital Subscriber Loop that allows high-frequency data communication between a home and a local telephone exchange. An xDSL modem operates at speeds of 400 Kbps, which is eight times faster than a 56-Kbps modem. The telephone company needs to install equipment at their branch exchange to communicate with this modem, so the service will not be available in some areas.

Networking is an in-home signaling scheme that can be used to interconnect multiple PCs in a single home. These networking signals do not propagate to the telephone company branch exchange (in fact, they are blocked from doing this to preserve your privacy), so no equipment needs to be installed there. The HomeRun consortium has been formed to standardize networking implementations.

All three services can operate on the same telephone wires at the same time with no interference between the services. An xDSL modem can be used at the same time as voice telephone calls—you effectively get your telephone line back!

An xDSL Modem Reference Design

The hardware connection to a POTS line to support xDSL is simplified following the introduction of purpose-built components. Figure 15-2 shows a block diagram of an xDSL modem using the Analog Devices chipset.

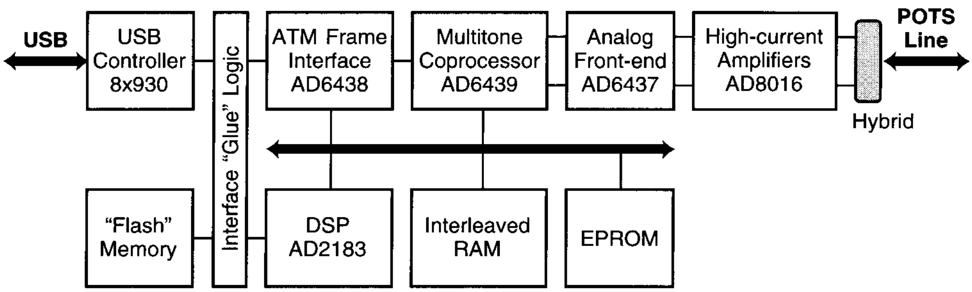


Figure 15-2. Block diagram of an xDSL modem

The design uses two microcontrollers that share the modem task. A Digital Signal Processor (AD2183 in the figure) is responsible for the analog side of the interface and includes high-output current amplifiers (AD8016) to drive the POTS line. Several supporting components (AD6437/8/9) construct outgoing ATM frames and decode incoming ATM frames that are processed by an Intel 8x930AD microcontroller.

The 8x930 handles modem initialization, configuration, and the high-speed, bulk data transfers to/from the PC host. The modem enumerates as a Communications Class Device, and the operating system uses an NDIS 5 Miniport Driver to communicate with the modem (Figure 15-3). Standard TCP/IP and IPX/SPX drivers included with the Windows 98 operating system allow the modem to be treated just like a network connection, so no other drivers are required.

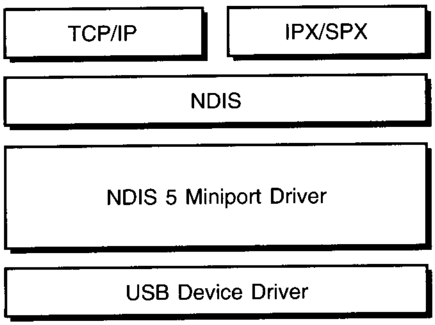


Figure 15-3. NDIS 5 miniport driver enables networking software

Figure 15-4 shows a photograph of the current example design.



Courtesy of Intel Corp., Architecture Development Laboratory.

Figure 15-4. Example xDSL modem design

Home Networking Over Phone Wires

The Home Phoneline Networking Alliance (HomePNA) was formed with the objective of quickly creating a de facto standard and widely available in-home networking solution that leverages the existing phoneline. The HomePNA has two goals: to endorse an initial available technology at 1 Mbps that meets all of the above criteria, and to rapidly deliver a roadmap to higher performance and functionality that is backward-compatible with the existing 1-Mbps solution. This technology promises to bring the power of networking to home users for the first time in a way that is inexpensive and easy to use.

Figure 15-5 shows a possible home network diagram, and the phone line to PC connection is a natural one for USB.

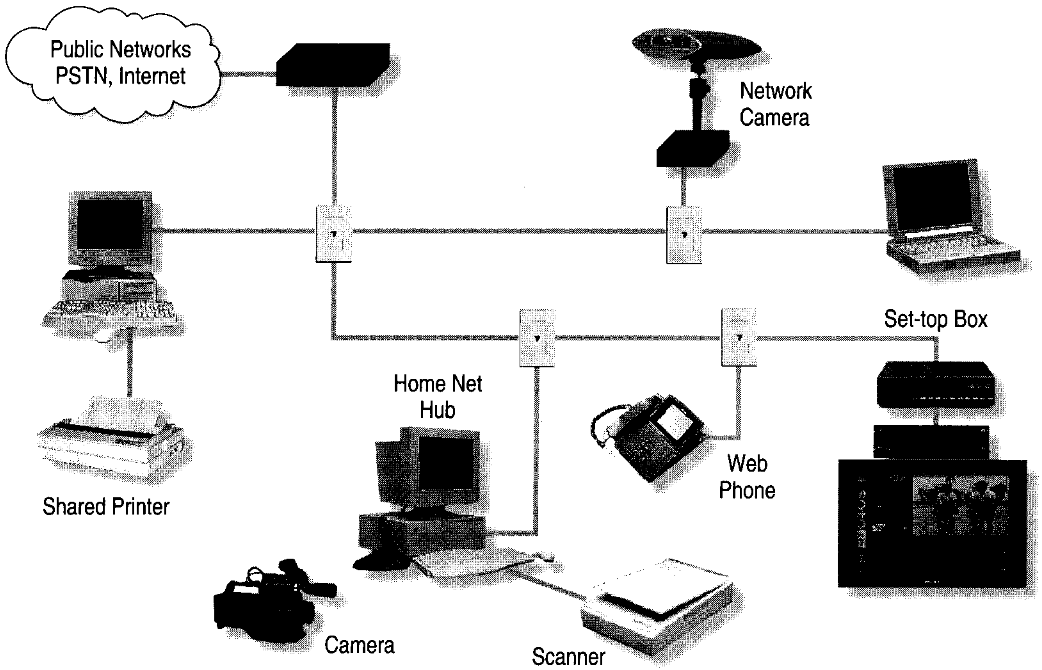


Figure 15-5. Possible home network using phone wires

Intel has introduced a single-chip Phoneline/Ethernet LAN controller that can be designed inside a PC host or in a USB dongle with a supporting microcontroller such as the Intel 8x930 (Figure 15-6). The PC host will use its included networking support, so no more software will be required. Full peer-to-peer networking will make it straightforward for multiple home PCs to share resources such as disk drives, printers, and scanners.

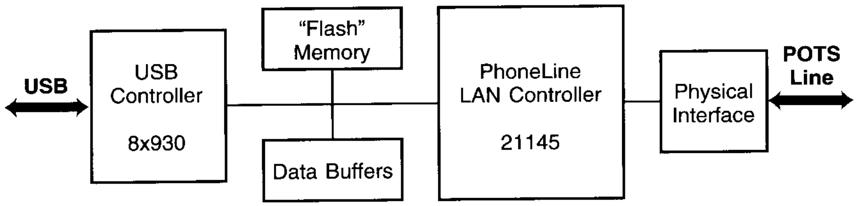


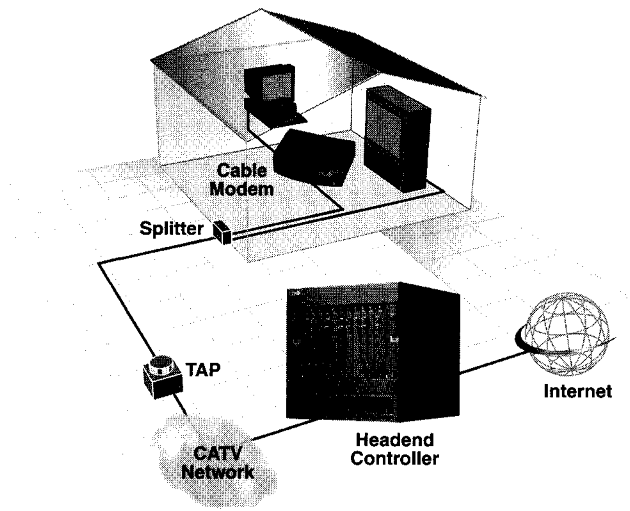
Figure 15-6. Possible USB-to-phoneline networking dongle

A NEW LOOK AT CABLE COAX

The cable coax coming into your home represents a very large-capacity data pipe. Most cable TV systems are analog at this time, and the designs in this section interface directly to this signaling scheme. Some newer installations are **digital cable**, and these designs are covered in the following "Digital Broadcast and the PC" section. Two aspects of using cable coax will be covered: a cable modem design that allows a very high bandwidth connection to the Internet; and an in-house, all-functions, network currently in user trials by Peracom Corporation.

Cable Modem Example

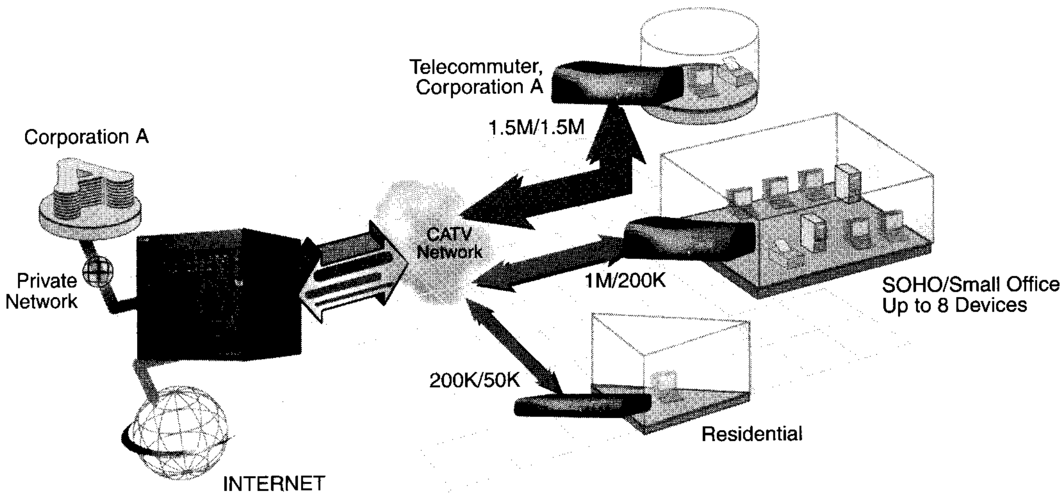
Although they share a similar name, a telephone modem and a cable modem are quite different in operation. A cable modem is a modem in the true sense of the word—it MOdulates and DEModulates signals. But the similarity ends there, because cable modems are practically an order of magnitude more complicated than their telephone counterparts. Cable modems can be part-modem, part-tuner, part-encryption/decryption device, part-bridge, part-router, part-NIC card, part-SNMP agent, and part-Ethernet hub. In the typical home installation shown in Figure 15-7, a splitter is used to connect both a cable modem and a TV to the same coax cable. The cable modem sends and receives data in two slightly different ways. In the downstream direction, the digital data is modulated and then placed on a 6-MHz television carrier, somewhere between 42 MHz and 750 MHz. This signal can be placed in a 6-MHz channel adjacent to TV signals on either side without disturbing the cable television video signals. The upstream channel, also known as the reverse path, is transmitted between 5 and 40 MHz.



Courtesy of COM21, Inc.

Figure 15-7. Cable modem and TV signals use the same coax

Figure 15-8 shows the bigger picture that includes the cable company's headend equipment and their very high-speed connection to the Internet. A cable modem is always on, so there is no need to dial up or otherwise wait for a connection. The modem's speed is between 200 Kbps and 2 Mbps—large files that can take up to 10 minutes to download using a telephone modem at 28.8 Kb will download in about **8 seconds** using a cable modem. Once you've used one, you will never want to give it back!



Courtesy of COM21, Inc.

Figure 15-8. Cable modem/TV system overview

USB is an ideal connection for the cable modem to the PC because it is easy, supports the data rate requirements, and creates a good demarcation between "my PC" and "the cable company's modem." Earlier versions of cable modem were implemented as a PCI plug-in card, and a skilled technician was required to install and set up a user's environment—quite an expensive procedure. So let's look inside a cable modem to see how it is built. Figure 15-9 shows a block diagram of a typical cable modem; this example is based on Broadcom's chipset.

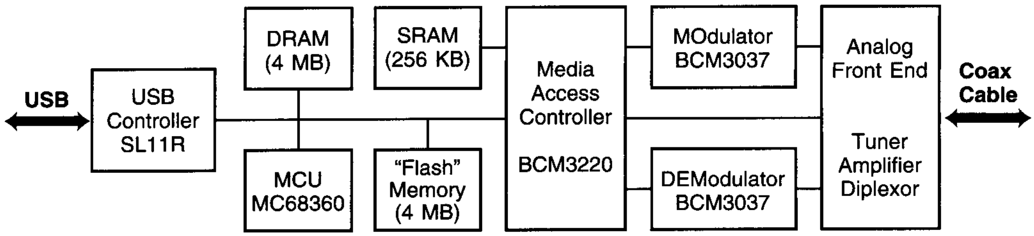
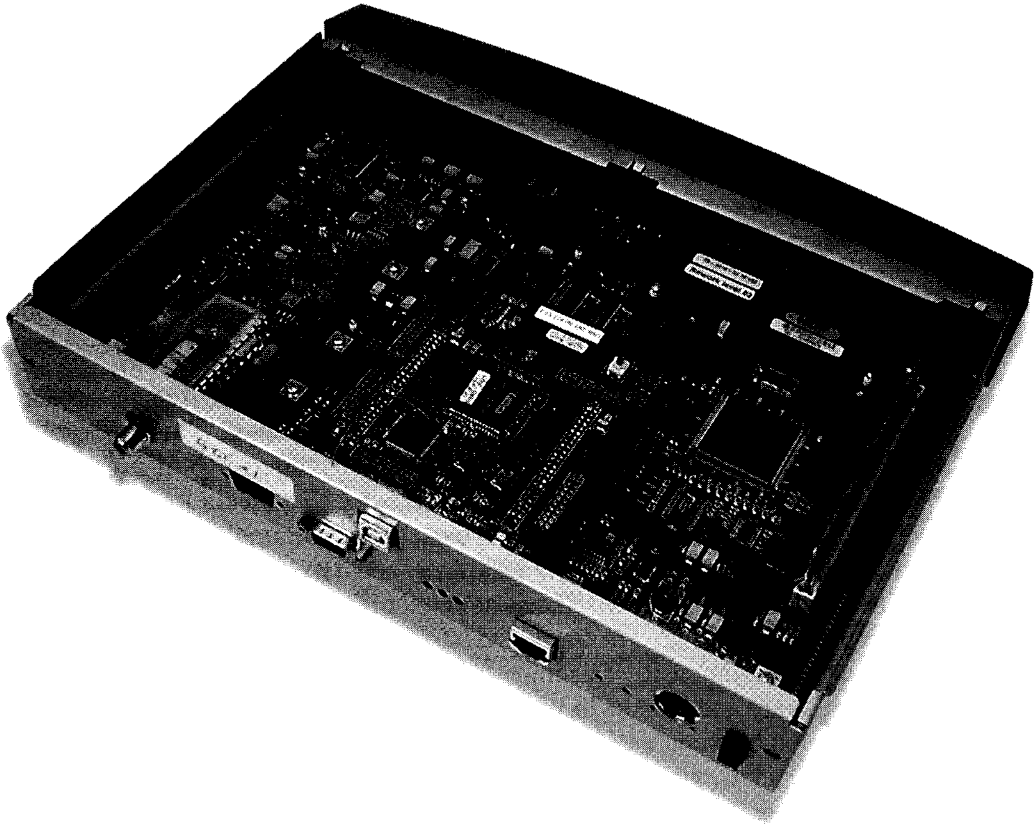


Figure 15-9. Block diagram of a cable modem

The incoming modulated data (downstream) in 50-to-850-MHz frequency band is amplified and converted to baseband by the front-end tuner. The baseband signal is digitized and processed by a QAM demodulator/error corrector (BCM3116). The corrected data stream is presented to a Media Access Controller (BCM3220 in the figure) that parses the incoming data frames and extracts timing information, control messages, and data packets, and stores them in high-speed shared memory. The system microcontroller (MC68360EN) accesses this data in shared memory and performs transactions that are requested by the headend equipment. This microcontroller runs embedded code stored in the flash memory and uses its local DRAM for data and code. All communication between the microcontroller and other units in the system occurs over the system bus, which provides a 32-bit data path to DRAM and 8/16-bit interface to other units. The microcontroller forwards IEEE 802.3 data packets that are directed to the PC host from SRAM into the USB controller (SL11R). The USB controller manages a high-speed data connection to the PC host, the transfer occurring, in this example, as many bulk data transfers.

In addition to supporting the standard device requests on Endpoint 0, the cable modem also supports Communications Device Class requests. Specifically, the modem supports the Ethernet Networking Control Model as identified by the subclass code. The PC host software is built around an NDIS Miniport driver (just as the xDSL modem was) with a DOCSIS-compliant (Data-Over-Cable Service Interface Specification) interface above that. For an example design, see Figure 15-10.

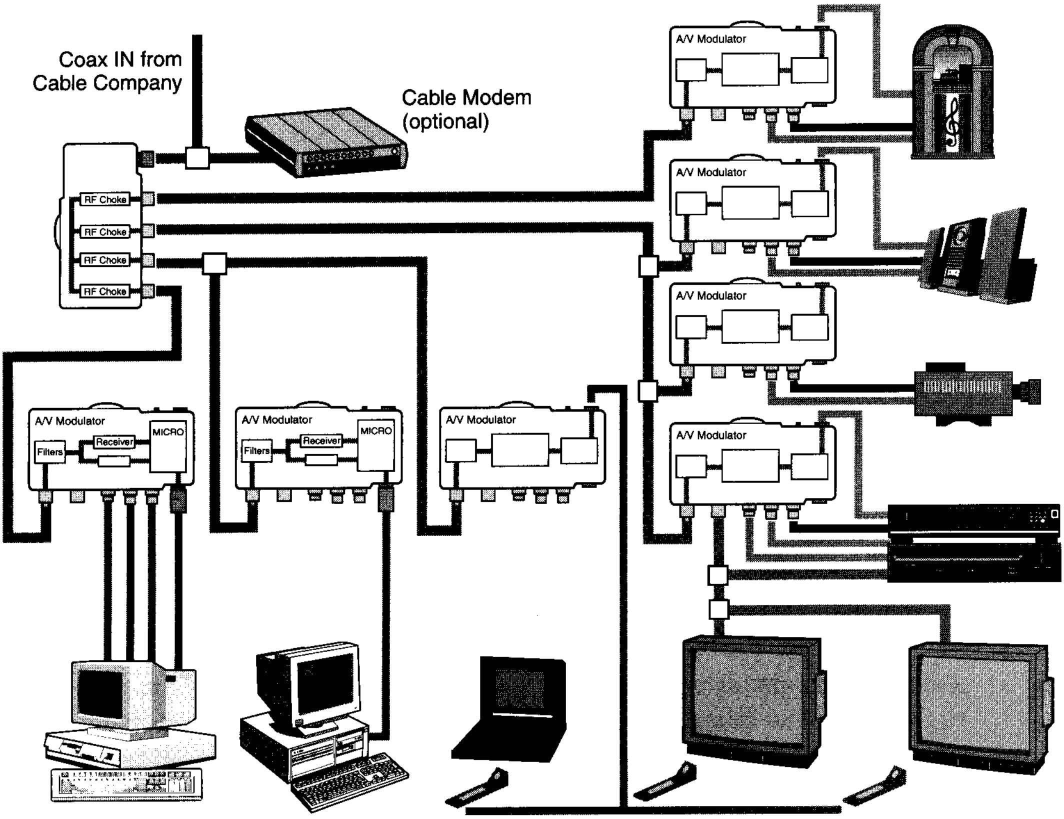


Courtesy of Architecture Development Laboratory, Intel Corp.

Figure 15-10. Example design implementation

Cable Networking Example

This section uses Peracom's HomeConnex product as an example of the ease of use that can be created with USB technology. HomeConnex uses the existing cable infrastructure in a home and adds capability via intelligent slave units, called casters, as shown in Figure 15-11.



Courtesy of Peracom Networks, Inc.

Figure 15-11. Peracom's HomeConnex networking product

PC Caster Overview

Each computer connects to a coaxial cable through a PC modulator that enables print and file sharing over a Computer Data Highway. The PC Modulator also creates an in-home channel of your broadcast-enabled PC's screen, making it much easier for DVD-equipped computers to display to any TV in the house. With an IR keyboard and RF mouse, you can control your computer from any room in the house. The computer can send IR codes over the network, as well as control which IR signals go where, so your entire home network can be controlled by the PC.

Media Caster Overview

An Audio/Video Modulator allows you to connect any audio/video device to the coaxial cable network in your home, and send the output to every TV set in your house. No special wiring is needed: simply plug the system side of the modulator into an existing Cable TV (CATV) outlet in any room of your house; then plug the TV side into your television; and finally, connect your audio/video device, such as a VCR, into the RCA jacks. The A/V modulator will be broadcast to every TV in your home, on its own channel (that you choose). This will work with any device with A/V outputs, including DSS, DVD, Laser Disc, VCR, Audio Jukebox, and Security Cameras. The modulator is equipped with an IR port as well, which allows you to control any device that uses infrared from anywhere in your home.

Cable Caster Overview

A distribution unit distributes Cable TV, in-home TV channels, a Computer Data Highway, and an Infrared Highway to multiple coax drops. The distribution unit couples upstream TV to the other coax drops connected to the bidirectional F connectors. The distribution unit also amplifies signals from the cable company or TV antenna for home distribution. There is a notch filter built into the distribution unit to prevent in-home channels and data networking channels from leaving the home.

The coax cable is the carrier of multiple independent channels (Figure 15-12). In addition to the expected cable TV channels, the system is designed for 16 in-home channels that can be created by a PC caster or an A/V caster. The coax also has a computer data "channel," an infrared "channel," and allocation for shared cable modem "channel."

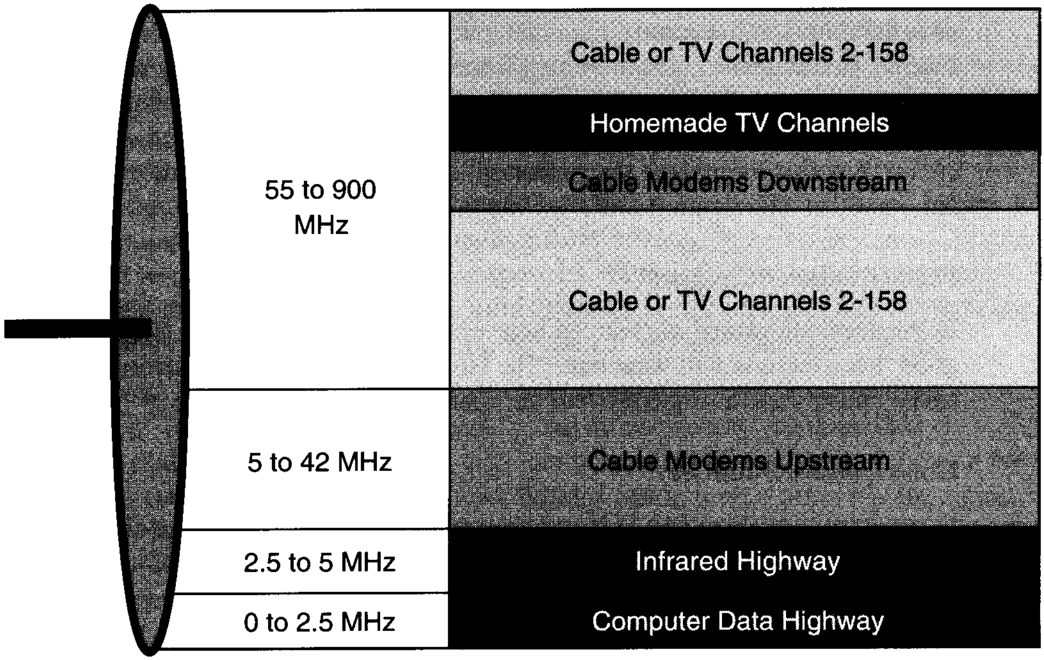


Figure 15-12. Bandwidth allocation on the HomeConnex network

Computer Data "Channel"

Multiple computers can share files and printers using the computer data "channel." The hardware connection to the PC is via USB, and the PC caster units communicate with each other over this computer data channel. The software used to transfer files and to print on remote printers is already built into the Windows operating system. An IrDA Data channel provides the file transfer mechanism over the infrared channel for laptop computers and IR-equipped desktop computers.

The Infrared "Channel"

Almost all consumer electronics devices are controllable via an infrared remote control. Unfortunately, they are all different, so you need *n* remotes for *n* devices. Multibrand and learning remotes have eased the coffee-table clutter of remotes, and Peracom's HomeConnex system can take us the next step. We now have a PC host in the control loop that gives the system a lot more flexibility and ease of use. Signals from any remote control can be directed to any receiver—you can even play your Sony PlayStation from **any** room by tuning into your PlayStation channel and using a Sony IR remote control. The PC host can also create IR control signals to stop and start VCRs or do whatever an old IR remote did.

Cable Modem "Channel"

In the previous section we learned how wonderful a cable modem is—it's so good that everyone will want one. The HomeConnex system lets every PC host in the house share a single modem transparently.

HomeConnex Summary

If you have more than two PCs and more than two televisions, you will discover that HomeConnex is an excellent **home** networking product. I emphasize the word home because we are interconnecting more than just the PCs—we are creating a whole-house video and sound system.

Projects that you have put off, such as a security camera over your yard or music on the deck, are very simple additions with this system. I've been using the product for several months now, and new uses are appearing all the time—I can view my son's PC screen from my office and feel comfortable that he is doing his homework. I can also see how well he is doing on his Sony PlayStation game. We now have a "Mom" channel that turns on every TV in the house, if they are not on, and allows my wife to rally the household for dinner.

The ease in which my laptop can connect into the home-networking, infrared channel from almost every room in the house means that I can be even more productive at home. System reconfiguration and upgrades are also easy because of the combination of USB and infrared peripherals.

This is the kind of science fiction system that George Jetson would have, and with USB it is available today.

Digital Broadcast and the PC

The availability of a high-speed digital data delivery mechanism integrated into the satellite direct broadcasting system creates a unique opportunity to design and introduce a new PC peripheral. This peripheral would receive the standard digital broadcast signal from a direct broadcast satellite (DBS) or a digital broadcast over the cable network, and would forward the demodulated digital stream to a PC. Because this digital data stream can carry data of any nature, the peripheral can be used with video, audio, and computer data broadcasts that, being received by PC, could be handled by software to create adequate presentation to the PC user. In the case of digital TV, the software will be able to decompress and display video on a PC screen. If audio data is filtered by the peripheral from the composite multichannel stream, the software can decompress and re-create audio and send it to speakers via the standard audio subsystem. High-speed data streams can be used to deliver a rich data content, and their use is limited only by the creativity of the content and application developer.

The development of digital broadcasting services evolves at a surprisingly fast pace. From the technical perspective, the infrastructure for digital data distribution is already in place, with more capacity and features to follow in the near future. The existing broadcast systems that appeared primarily for digital TV are built on well-defined standards and specifications, which are suitable for data broadcast as well. The international standards used for digital TV broadcasting provide a solid foundation for future interoperability.

This section introduces the design of a low-cost, stand-alone, USB-based, receiver designed to receive satellite broadcasts, demodulate the signal, filter appropriate data streams, and forward them to a PC host. The system uses transport streams and data formats, which are defined by the ISO 13818 (MPEG Transport stream) and the ETSI DVB-S specifications. The material is presented to give a flavor of the design; full details are available, under license, by contacting the author.

The receiver is one of the components of the broadcast system based on the DVB specifications. The whole system consists of the satellite headend, direct broadcast satellite(s), low-noise outdoor unit with a dish antenna and LNBF unit, the indoor receiver, and a PC with appropriate software (Figure 15-13).

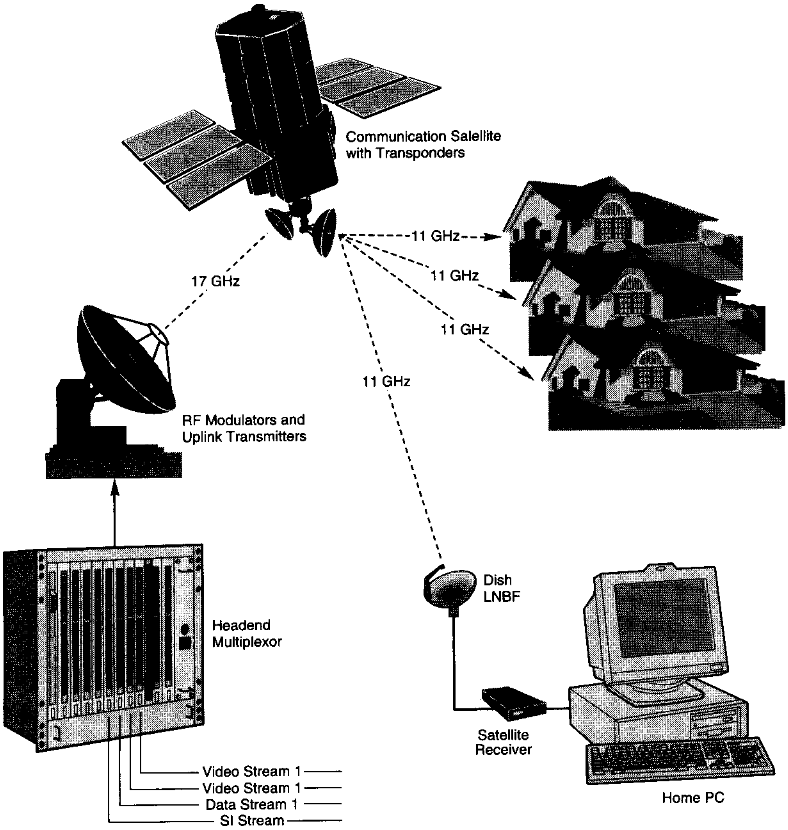


Figure 15-13. Direct broadcast satellite system

At the headend, TV and digital data streams are multiplexed, and an RF signal is modulated by the composite data stream. The uplink transmitter working in the 17.3-to-17.8-GHz frequency band sends the signal to the satellite, which is equipped with transponders. The transponder receives this uplink signal, converts its frequency to one of the Ku-band downlink channels, amplifies it, and sends it back to Earth with a directional antenna. There may be several transponders on the satellite, each tuned to a separate individual uplink and downlink frequency.

Each transponder can deliver up to 41-Mbps data streams that may contain multiple audio, video, and data channels. The downlink transmitters use circular polarization to reduce dependencies on the atmosphere conditions and simplify the receiver antenna alignment. When the 12-GHz Ku-band satellite signal reaches the dish antenna, it is first amplified by the low-noise microwave amplifier and then converted to the first intermediate frequency (950-to-2150 MHz). These functions are done by an integrated LNBF unit to simplify the transmission and further processing of the signal, and to reduce the overall noise level in the system. It is important, however, that all 22 transponder signals are present in the wide-band signal sent over coaxial cable from the dish to the satellite receiver. Figure 15-14 looks inside the satellite receiver.

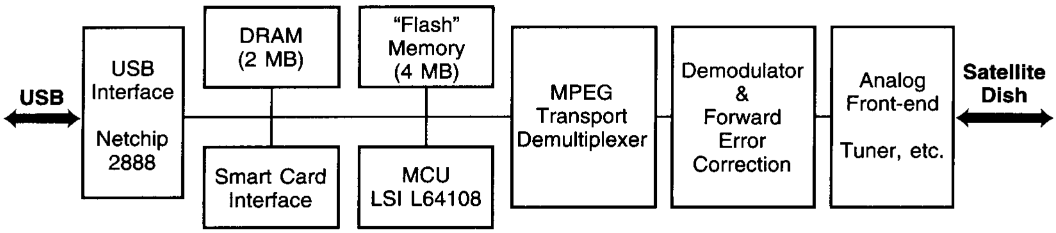


Figure 15-14. Block diagram of a satellite receiver

When the first IF signal reaches the tuner, it filters only one transponder's band and further amplifies and converts the signal into a second IF signal. After filtering, the IF signal is demodulated, and Forward Error Correction algorithms are applied. These techniques guarantee the reliable demodulation even in the presence of noise and RF interference. Unlike the conventional analog TV that uses a separate frequency channel for each TV program, multiple digital TV programs can be sent in the same multiplexed MPEG transport stream on the same carrier frequency. Therefore, before data can be viewed, the transport stream must be demultiplexed, and individual streams must be extracted for audio, video, and data.

The USB interface controls data exchange over the USB bus. It supports several logical data connections, and it is the only interface to the PC host. The biggest portion of the USB bandwidth is used for a unidirectional data stream from the MPEG transport demultiplexer. When the data is received by the USB port in the PC, it is processed first by the USB controller driver, which supports the physical USB interface, and then by the operating system that in turn dispatches the received packets to the applications to which they belong. For example, video and audio streams are directed to the MPEG player application, which decompresses the data and presents it on the PC screen.

CHAPTER SUMMARY

USB is a very convenient, high-speed pipe used to get data into and out of a PC host. This chapter has reviewed several methods of increasing the data bandwidth into the home (or small office!). In all cases the connection to the PC host was USB, which simplified each installation. Sharing information via an in-home network and having instant (no dial-up) access to the Internet will enable the whole family to be more efficient. Whether it's researching the Civil War for a school project, downloading large specifications, or high-speed surfing, USB can deliver the capability of easy-to-use operation.

Important too for the telecommunications applications discussed in this chapter is the demarkation between "my PC" and "your communications equipment." Because the connection is a standard USB cable, the equipment at each end can be independently tested to isolate "my problem" from "your problem."

USB is already having a positive effect on the PC host and a growing number of applications. With the help and information from this book, you too will be able to make an impact! Good luck.